VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Neuronová síť typu Flexible Neural Tree

Flexible Neural Tree

2015 Jiří Hanzelka

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Jiří Hanzelka**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Neuronová síť typu Flexible Neural Tree
Flexible Neural Tree

Zásady pro vypracování:

Cílem práce je implementovat Flexible Neural Tree (FNT) v prostředí HPC. Stěžejními částmi práce je identifikace částí kódu s vysokou časovou složitostí, návrh paralelního zpracování pomocí vypočetního klastru s využitím technologie MPI. Práce bude obsahovat následující části:

1. Rešerše současného stavu poznání v oblasti FNT, rešerše dostupných implementací FNT.
2. Identifikace míst s vysokou časovou složitostí. Návrh paralelního zpracování.
3. Objektová analýza navrhovaného řešení. Implementace tohoto řešení.
4. Testy a experimenty s implementovanou FNT.

Seznam doporučené odborné literatury:

[1] Yuehui Chen, Ajith Abraham: Tree-Structure based Hybrid Computational Intelligence, Springer, 2010

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **doc. Mgr. Jiří Dvorský, Ph.D.**

Datum zadání: 01.09.2014
Datum odevzdání: 07.05.2015

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Souhlasím se zveřejněním této diplomové práce dle požadavků čl. 26, odst. 9 *Studijního a zkušebního řádu pro studium v magisterských programech VŠB-TU Ostrava.*

V Ostravě 7. května 2015 .....

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 7. května 2015 .....

Rád bych na tomto místě poděkoval vedoucímu diplomové práce panu Doc. Mgr. Jiřímu Dvorskému, Ph.D., protože bez něj by tato práce nevznikla.

Abstrakt

Tato práce pojednává o paralelním řešení neuronové sítě typu Flexibilní Neuronový Strom. Výsledkem práce je naimplementovaná knihovna, která obsahuje paralelní verze algortimů pro nalezení struktury a parametrů neuronového stromu. Paralelní verze algoritmů využivají rozhraní MPI. Tato knihovna je naimplementována v prostředí .NET tak, aby mohla být použita pomocí Mono virtuálního stroje i na linuxu.

Klíčová slova: Flexibilní Neuronový Strom, MPI, .NET, Mono

Abstract

This thesis is about parallel approach of neural network called Flexible Neural Tree. Result of this thesis is implemented library which contains parallel versions of algorithms for structure and parameter optimization of neural tree. Parallel versions of algorithms uses MPI interface. This library is implemented in .NET enviroment in way to be used by Mono virtual machine even on linux system.

Keywords: Flexible Neural Tree, MPI, .NET, Mono

Seznam použitých zkratek a symbolů

FNT	–	Flexible Neural Tree
PSO	–	Particle Swarm Optimization
HPC	–	High Performance Computing
MPI	–	Message Passing Interface
GP	–	Genetic Programming
GA	–	Genetic Algorithm
SA	–	Simulated Annealing
AP	–	Ant Programming
ACO	–	Ant Colony Optimization
PIPE	–	Probabilistic Incremental Program Evolution
HONN	–	Higher Order Neural Network
DE	–	Differential Evolution

Obsah

Seznam tabulek

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

Neuronové sítě obecně byly inspirovány lidským mozkem pro pochopení jeho funkčnosti a možnosti využití na problémy neřešitelné konvenčními metodami. Z počátku bylo vytvořeno několik modelů neuronů, nejdůležitější byl *Perceptron* vytvořený McCulochem a Pittsem. Jednalo se o binární neuron, který dával výstupy 0 a 1. Nevědělo se však jak nastavit váhy neuron, aby tento perceptron dával požadované výstupy na dané vstupy [10]. O pár let později přišel Donald Hebb s algoritmem nazvaným Hebbovo učení, který dokázal nastavit váhy tohoto perceptronu. Následně byl perceptron upraven pro vydávání reálných výstupů. Tyto perceptrony byly spojovány do vícevrstvých sítí. Na tyto sítě však musel být vyvinut nový algoritmus učení. S tím se přišlo až později a tento algoritmus byl pojmenován jako metoda *Back-propagation*, tento algoritmus je dosud jeden z nejpoužívanějších kvůli své rychlosti učení [8].

Problémem těchto neuronových sítí je zvolení správné struktury. Struktura je totiž závislá na řešeném problému, malá síť se není schopna naučit všechny vzory trénovací množiny a u velké je problém dlouhá doba učení. K tomu počty neuronů v jednotlivých vrstvách ovlivňují schopnost zobecňování neuronové sítě. Jeden z modelů neuronový sítí, který je schopen vyhledat vhodnou strukturu sítě, se nazývá *Flexibilní neuronový strom* (FNT), anglicky *Flexible neural tree*. Tímto model se budu zabývat v této práci.

Cílem této práce tedy bude vytvoření knihovny tohoto modelu, která bude moci být použita pro řešení různých problémů. Z důvodu časové složitosti tohoto algoritmu, bude navržena jeho paralelizace v prosředí MPI. Testy budou provedeny na novém superpočítači Anselm na VŠB Technické Univerzitě v Ostravě. Testy budou měřit škálovatelnos navrhnutého algoritmu a poté samotné použití této knihovny na různé problémy.

2 Flexibilní neuronový strom

2.1 Vznik

V roce 1997 vytvořili autoři Byoung-Tak Zhang, Peter Ohm, Heinz Mühlenbein studii o řídkých neuronových stromech [3]. Z této studie později vznikly Flexibilní neuronové stromy. V této studii autoři přišli se stromovou strukturou neuronové sítě, která má mít lepší vlastnost zobecňování než HONN sítě. V těchto sítích je potenciál neuronu vypočítán jako suma součinů vah a vstupů vedoucích do neuronu viz následovně:

$$z = \sum_{j=1}^{m} w_j x_j \tag{1}$$

Nebo je vypočítán jako produkt součinů vah a vstupů vedoucích do neuronu:

$$z = \prod_{j=1}^{m} w_j x_j \tag{2}$$

Tabulka 1: Použité notace v této práci

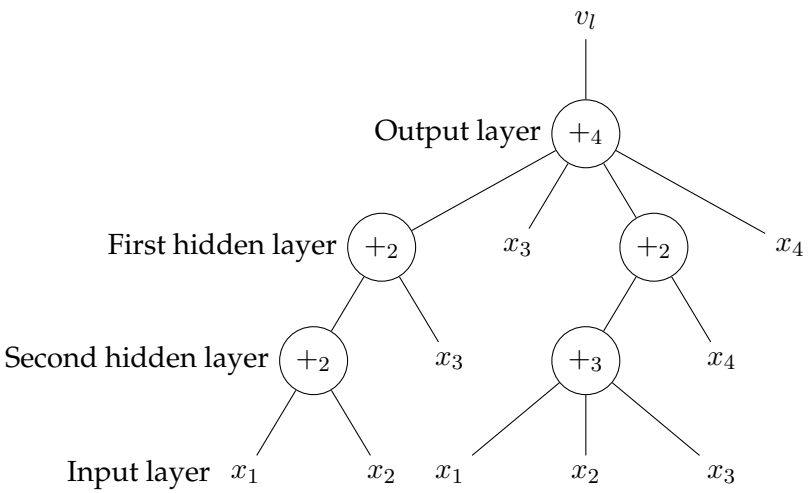
Symbol	Popis
M	maximální počet vstupů do neuronu
m	počet vstupů do neuronu
T	trénovací množina, $T = \{t_1, t_2, ..., t_P\}$
P	počet vzorů trénovací množiny
t_p	p-tý vzor trénovací množiny, skládá se ze vstupního a výstupního vektoru $t_p = \{\vec{I_p}, \vec{O_p}\}$
$\vec{I_p}$	vstupní vektor p-tého vzoru trénovací množiny, kde $\vec{I_p} \in \mathbb{R}^K$, kde K je dimenze vstupních vektorů
$\vec{O_p}$	výstupní vektor p-tého vzoru trénovací množiny, kde $\vec{O_p} \in \mathbb{R}^L$, kde L je dimenze výstupních vektorů
$o_{p,l}$	l-tá složka výstupního vektoru $\vec{O_p}$ a p-tého vzoru, kde $l = 1, 2, ..., L$ a $p = 1, 2, ..., P$
$v_l(t_p)$	výstup l-tého neuronového stromu pro vzor t_p
z	potenciál neuronu
w_j	j-tá vstupní váha neuronu, kde $j = 1, 2, ..., m$
x_j	j-tý vstup neuronu, kde $j = 1, 2, ..., m$
a	parametr aktivační funkce
b	parametr aktivační funkce
y	výstup neuronu
$rand$	náhodně vygenerované reálné číslo v intervalu $\langle 0, 1)$

J	velikost populace
D	dimenze řešeného problému optimalizačního algoritmu
$\vec{s}$	vektor aktuální pozice, kde $\vec{s} \in \mathbb{R}^D$
$\vec{s_0}$	vektor původní pozice, kde $\vec{s_0} \in \mathbb{R}^D$
$s_{i,j}(t)$	*j*-tá složka aktuální pozice *i*-tého jedince v čase *t*
$r_{i,j}(t)$	*j*-tá složka rychlosti *i*-tého jedince v čase *t*, kde $j = 1, 2, ..., D$
c_0	parametr setrvačnosti rychlosti, s každou iterací snižuje rychlost jedince, $c_0 \in \mathbb{R}$
c_{start}	počáteční hodnota setrvačnosti
c_{end}	koncová hodnota setrvačnosti
c_1, c_2	parametry ovlivňující směr rychlosti
$pBest_{i,j}$	*j*-tá složka nejlepší pozice *i*-tého jedince
$gBest_j$	*j*-tá složka nejlepší pozice
it	aktuální iterace
$maxIterations$	maximální počet iterací optimalizačního algoritmu
$f(\vec{s})$	zdatnost pro pozici $\vec{s}$
$temp$	aktuální teplota, reálné číslo ve zvoleném intervalu
ns_j	*j*-tá složka šumového vektoru
$s_{best,j}$	pozice nejlepšího jedince
F	mutační konstanta DE algoritmu, reálné číslo v intervalu (0,1)
r	náhodně vygenerované přirozené číslo v intervalu $\langle 1, J \rangle$

K tvorbě struktury používají metodu *Genetické programování* (GP) a k učení *Genetický algoritmus* (GA). Z důvodu náročnosti učení neuronových stromů je učena jen část populace. K tomu zpočátku jsou stromy učeny kratší dobu a až později se doba učení prodlužuje. Přišli i s řešením nekontrolované expanze struktur (bloating), kdy začaly s každou generací optimalizace struktury vznikat příliš velké neuronové stromy tím, že penalizovali větší struktury. Proto menší struktura se stejnou chybou má větší zdatnost než větší struktura se stejnou chybou.

Později se začali tímto tématem zabývat další vědci. Bylo vytvořeno několik publikací na Flexibilní neuronové stromy, které zjednodušily Řídké neuronové stromy [6, 7]. Později se tyto články sjednotily do práce Tree-Structure Based Hybrid Computational Intelligence: Theoretical Foundations and Applications [1]. Oproti řídkým neronovým stromům neurony provádí pouze sumaci vstupů a vah. Dále algoritmus učení stromů nebyl použit na část populace, ale jen na nejlepšího jedince který po několika generacích optimalizace struktury vznikl.

V roce 2011 vyšla publikace o paralelním řešení FNT algoritmu s využitím MPI pro superpočítače [5]. Pro optimalizaci struktury byla použita metoda PIPE a pro optimalizaci parametrů metoda PSO. Porovnávají zde dva přístupy k paralelizaci algoritmu.



Obrázek 1: Flexibilní neuronový strom

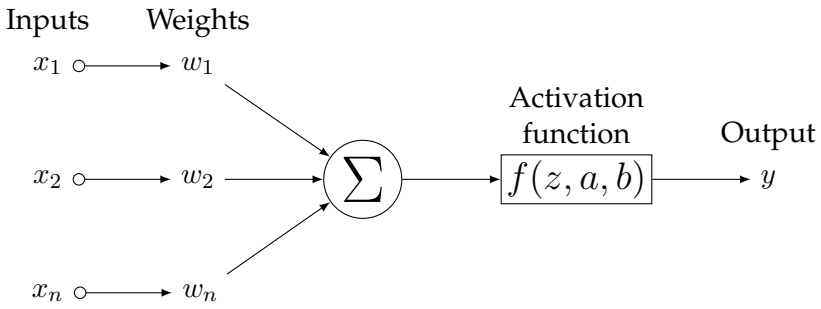
jeden je nazván *Phase parallel model* a druhý *Working pool parallel model*. Druhý přístup byl efektivnější. V obou přístupech ale řeší paralelizaci výpočtu chyby neuronových stromů, protože pro dostatečně velkou trénovací množinu a populaci jedinců se jedná o nejnáročnější část. První model rozděluje populaci na stejně velké podpopulace, které jsou následně poslány k ohodnocení všem procesům. Čas ohodnocení je tedy roven nejpomalejšímu procesu, protože se jedná o synchronní verzi algoritmu. Druhý model používal asynchronní přístup, kdy rozesílal práci volným procesům a dosahoval podle testů lepší efektivity.

2.2 Popis

Flexibilní neuronový strom, viz obr. 1, je speciální typ dopředné neuronové sítě, který má nepravidelnou strukturu a pouze jeden výstup. Pokud potřebujeme více výstupů tak neuronová síť obsahuje neuronový strom pro každý výstup trénovací množiny. Výstup takového neuronového stromu je možné vypočítat rekurzivně metodou průchodu do hloubky. Matematicky se jedná o množinu terminálních symbolů T (vstupů) a funkčních symbolů F (neuronů nebo taky flexibilních neuronových operátorů):

$$TREE = F \cup T = \{+_1, +_2, ..., +_M\} \cup \{x_1, ...x_K\} \tag{3}$$

Vstupů může být libovolné množství a mohou se i opakovat. Vstupy mohou vést přes vrstvu a dokonce i rovnou k výstupnímu neuronu. Stromová struktura této sítě umožňuje použití biologicky inspirovaných algoritmů pro tvorbu stromů. Mezi tyto algoritmy patří *Genetické programování* (GP), *Mravenčí programování* (AP) nebo algoritmus *Pravděpodobnostní inkrementální programové evoluce* (PIPE). Tyto algoritmy umožňují nalezení vhodné strukrury neuronové sítě pro trénovací množinu. Kromě hledání vhodné struktury této neuronové sítě, je potřeba zároveň optimalizovat i parametry sítě (učit neuronovou síť).



Obrázek 2: Flexibilní neuronový operátor

Algoritmy k učení sítě mohou být metoda *Back-propagation* (BP), *Genetické algoritmy* (GA), *Rojení částic* (PSO) nebo *Simulované žíhání* (SA). U flexibilních neuronových stromů se muselo přistoupit ke kompromisu. Ten spočívá v tom, že z důvodu vysoké výpočetní složitosti se neučí všechny stromy v nové generaci, ale nechá se algoritmus běžet určitý počet generací bez učení stromů a poté se vybere nejlepší strom z aktuální generace a u něj dojde k optimalizaci parametrů učením. Tento cyklus, kdy se hledá vhodná struktura a poté učí nejlepší strom se nazývá epocha a opakuje se, dokud není nalezeno požadované řešení nebo dokud počet epoch nepřekročí navolenou hodnotu [1].

2.2.1 Trénovací množina

Trénovací množina je složena ze vzorů. Každý vzor je potom složen z vektoru vstupů a vektoru požadovaných výstupů. Neuronová síť se musí tyto vzory naučit, tzn. musí se nastavit váhy a parametry aktivačních funkcí tak, aby pro daný vstup síť vydala požadovaný výstup. Trénovací množina se normalizuje do reálného intervalu $\langle 0, 1\rangle$ nebo $\langle -1, 1\rangle$, z důvodu používaných aktivačních funkcí [8]. Trénovací množinu lze matematicky popsat následovně:

$$T = \{\{\vec{I_1}, \vec{O_1}\}, \{\vec{I_2}, \vec{O_2}\}, \ldots, \{\vec{I_P}, \vec{O_P}\}\} \tag{4}$$

2.2.2 Flexibilní neuronový operátor

Jedná se o neuron FNT neuronové sítě viz obr. 2. Ten může mít několik vstupů, minimálně však jeden a jeden výstup. Jako u dopředných neuronových sítí je vstup zesílen nebo zeslaben pomocí vah. Celkový potenciál z neuronu $+m$, kde m udává počet vstupů do neuronu, se vypočte viz následující rovnice.

$$z = \sum_{j=1}^{m} w_j x_j \tag{5}$$

Tento signál je poté upraven pomocí aktivační funkce, která má nastavitelné parametry a a b, které dále upravují výstup, parametr a určuje strmost funkce, parametr b slouží jako práh. Existuje mnoho typů aktivačních funkcí, jako jsou například:

- Sigmoidální funkce

$$y = f(z, a, b) = \frac{1}{1 + e^{\frac{-(z-b)}{a}}} \tag{6}$$

- Hyperbolický tangens

$$y = f(z, a, b) = \frac{e^{\frac{2(z-b)}{a}} - 1}{e^{\frac{2(z-b)}{a}} + 1} \tag{7}$$

- Gaussova funkce

$$y = f(z, a, b) = e^{(-(\frac{z-b}{a})^2)} \tag{8}$$

2.2.3 Funkce zdatnosti

Aby bylo možné určit, která neuronová síť je pro danou trénovací množinu lepší než jiná, je nutné nějakým způsobem ohodnotit tyto neuronové sítě. K tomuto slouží funkce zdatnosti. U neuronových sítí je použita chyba sítě jako zdatnost.

Je několik typů výpočtu chyby sítě. Tyto funkce sčítají chyby jako rozdíl mezi požadovaným výstupem a výstupem neuronového stromu pro každý vzor trénovací množiny a dále jej upravují. Nejčastěji se používají následující dva typy:

- MSE (Mean Square Error)

$$MSE = \frac{1}{P} \sum_{p=1}^{P} (v_l(t_p) - o_{p,l})^2 \tag{9}$$

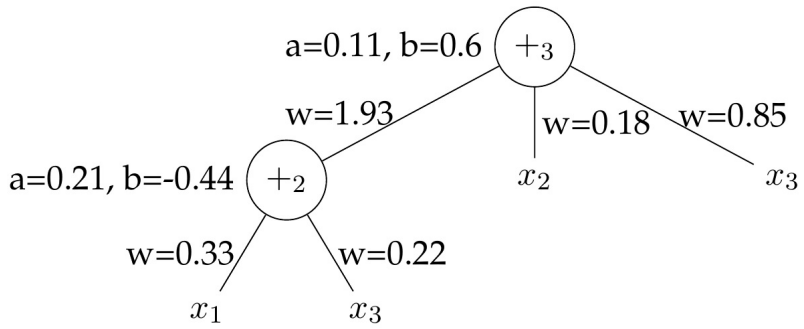
- RMSE (Root Mean Square Error)

$$RMSE = \sqrt{MSE} \tag{10}$$

2.2.4 Kódování struktury

V diplomové prácí Pavla Piskoře [2] byla struktura neuronového stromu kódována jako matice spojů mezi neurony. Nevýhoda tohoto přístupu byla nutnost odhadnout velikost této matice před samotným vyhledáváním řešení. Pokud byla matice moc velká, bylo vyhledávání pomalé, pokud byla malá, tak řešení nemohlo být dostatečně přesné.

Proto jsem použil jiné kódování. Díky stromové struktuře je možné kódovat strukturu stromu do řetězce proměnné délky metodou průchodu do hloubky. Kromě struktury stromu jsou do tohoto řetězce zakódovány i parametry jako jsou váhy spojů a parametry aktivační funkce a a b. FNT je tedy jednoznačně určen pomocí tohoto řetězce nazývaného DNA, tento název je zvolen podle biologie, kde je většina živých organismů podobně definována pomocí *deoxyribonukleové kyseliny* tedy DNA. Jako DNA řetězec pro algritmus



Obrázek 3: Neuronový strom s parametry

GP by byl neuronový strom z obr. 3 zakódován následujícím způsobem viz rovnice 11. Bez parametrů by to bylo způsobem viz rovnice 12.

$$DNA = \text{+3a0.11b0.6w1.93+2a0.21b-0.44w0.33x1w0.22x3w0.18x2w0.85x3} \quad (11)$$

$$DNA = \text{+3+2x1x3x2x3} \quad (12)$$

Tento způsob řešení má výhodu, že nemusí být specifikována maximální délka DNA řetězce. Další výhodou je, že v DNA nevznikají smyčky a různé chyby při optimalizaci struktury jako v případě kódování do matice spojů. Nevýhoda může být složitější práce s řetězci při operacích křížení a mutace u GP a jiných algoritmů pro optimalizaci struktury.

2.3 Optimalizace struktury

Struktura neuronové sítě významně ovlivňuje přesnost neuronové sítě na daná data, jež je schopna se neuronová síť naučit. U flexibilních neuronových stromů je možné použít k nalezení téměř optimální struktury algoritmy, jako jsou Genetické programování (GP), Propabilistic Incremental Program Evolution (PIPE), Ant Programming(AP) a jiné. V této práci se budu zabývat jen Genetickým programováním, protože jsem tento algoritmus použil k optimalizaci struktury FNT.

2.3.1 Genetické programování (Genetic programming)

Genetické programování navrhl John Koza. Je to modifikovaná verze genetického algoritmu, který je popsán později. Vychází z evoluce popsané Charlesem Darwinem. Liší se ale v reprezentaci jedinců [1, 9]. V GP je jedinec reprezentován stromovou strukturou, tím pádem strom zakódovaný do řetězce není omezen fixní délkou. GP ma svůj název podle toho, že bylo vytvořeno k tvorbě programů, které je možné zakódovat do stromové

struktury. Algoritmus používá i stejné názvy řídících parametrů. Je zde maximální počet generací, po kterou může algoritmus běžet, počet jedinců, parametr ovlivňující mutaci. Je možné i v tomto algoritmu zavést elitismus stejně jako v GA pro rychlejší konvergenci. U neuronových stromů, je potřeba v tomto algoritmu zavést parametr penalizace větších neuronových stromů, co mají stejnou zdatnost jako stromy menší. Zabrání to nekontrolované expanzi velikosti neuronových stromů.

Pro flexibilní neuronové stromy je kódování použito viz rovnice 11. Díky tomuto kódování je nutné mít upravené operace křížení a mutace.

- **Křížení** - Tato operace probíhá tak, že po vybrání rodičů se zvolí náhodně jeden funkční uzel (neuron) u každého rodiče a tyto uzly jež představují podstromy se přehodí mezi rodiči, tímto způsobem vzniknou dva noví potomci.
- **Reprodukce** - Vybraný jedinec se přesouvá do nové generace stejně jako u GA.
- **Mutace** - Po vytvoření potomka je šance že se provede jeho mutace, šance je ovlivňena řídícím parametrem algoritmu. U stromové struktury flexibilního neuronového stromu je několik typů mutace:

  1. Změna terminálu (vstupu) za jiný
  2. Změna všech terminálů za jiné
  3. Růst - nahrazení terminálu nově vygenerovaným stromem
  4. Prořezání - nahrazení neterminálu (neuronu) terminálem
  5. Volitelné
     (a) Přidání terminálu k náhodnému neterminálu
     (b) Odebrání terminálu náhodného neterminálu

2.4 Optimalizace parametrů

2.4.1 Rojení částic (Particle swarm optimization)

Optimalizace pomocí rojení částic (PSO) je hejnový algoritmus, který je inspirovaný hejny ptáků, ryb nebo spolupracujících lidí. Tento algoritmus využívá populaci částic, která prohledává daný prostor řešení. Zakladateli tohoto algoritmu byli Russel Eberhart a James Kennedy[9, 11]. Částice musí mít svou polohu v N dimenzionálním prostoru všech řešení. Pro použití v neuronových sítích jsou tyto pozice hodnoty vah a parametrů aktivačních funkcí. Dále má každá částice svoji rychlost, což je vektor o rozměru N, který v každé iteraci algoritmu mění aktuální pozici častice. Vektor rychlosti se počítá podle rovnice 13. Každá částice si pomatuje dosavad nejlepší řešení, které našla při procházením prostoru. Plus algoritmus musí vědět o dosavadním nejlepším řešení. Nová pozice částice se potom vypočítá podle rovnice 14. Pseudokód PSO algoritmu je ve výpisu 1.

Částice jsou navzájem ovlivňovány tím, že jsou přitahovány k částici, která představuje dosavadní nejlepší řešení, nebo jsou přitahovány ke svým doposud nejlepším nalezeným řešením. Směr kterým budou částice přitahovány je určen náhodně, ale je jej možné ovlivnit konstantamy *c1* a *c2*. Pokud je vyšší hodnota *c1* než *c2*, upřednostňuje částice směr k dosavad nejlepší pozici této částice. Pokud je hodnota *c2* vyšší než *c1*, tak to znamená, že častice bude častěji postupovak ke globálnímu minimu (doposud nejlepší nalezené pozici). Hodnoty *c1* a *c2* se volí nejčastěji kolem hodnot 2, záleží ale na problému, na který je algoritmus použit.

$$r_{i,j}(t+1) = c_0 \cdot r_{i,j}(t) + c_1 \cdot rand \cdot (pBest_{i,j} - s_{i,j}(t)) + c_2 \cdot rand \cdot (gBest_j - s_{i,j}(t)) \quad (13)$$

$$s_{i,j}(t+1) = s_{i,j}(t) + r_{i,j}(t+1) \quad (14)$$

$$c_0 = c_{start} - ((c_{start} - c_{end}) \cdot it)/maxIterations \quad (15)$$

```
Inputs:netParameters, algorithm parameters
Outputs:newNetParameters

public double[] Pso(double[] netParameters)
{
    set the first position to netParameters other randomize
    randomize velocities of all particles
    for(int it = 0; it < maxIterations; it++)
        c0 = cStart − ((cStart − cEnd) ∗ it)
        for (int i = 0; i < particlesCount; i++)
            for (int j = 0; j < dimension; j++)
                r[i][j] = c0 ∗ r[i][j] + c1 ∗ rand ∗ (pBest[i][j] − s[i][j])
                                       + c2 ∗ rand ∗ (gBest[j] − s[i][j])
                s[i][j] = s[i][j] + r[i][j]
                fitness = Evaluate(s[i][j])
                if (fitness < pBest[i])
                    pBest[i] = r
                if (fitness < gBest)
                    gBest = r
    return newNetParameters
}
```

Výpis 1: Pseudokód PSO algoritmu

V neuronových sítích trvá nejdéle výpočet chyby (ohodnocení) neuronové sítě pro danou trénovací množinu. Proto je možné algoritmus zparalelizovat pomocí MPI tak, že populace může být rozdělena na subpopulace a ty jsou poslány na ostatní procesy včetně master procesu s rankem 0. Master proces tedy provádí aktualizace rychlostí částic a jejich pozic a aktualizuje doposud nejlepší řešení pro jednotlivé částice *pBest* a nejlepší řešení pro všechny *gBest*. Slave procesy provádějí včetně master procesu výpočet chyby

sítě třeba RMSE viz rovnice 10. Ikdyž je tento algoritmus synchronní a čeká se na nejpomalejší proces, je možné dosáhnout slušné efektivity algoritmu. Této efektivity je možné dosáhnout pouze za podmínky, že počet částic je násobek počtu spuštěných procesů a částic je minimálně tolik co procesů. Dále je použit tento algoritmus na uzlech stejného výkonu. Tato paralelní verze algoritmu pro master proces je ve výpisu 2. Metoda *scatter* v MPI vrátí i master procesu část populace pro ohodnocení chyby, to umožňuje použití tohoto algoritmu i na jednom procesu. Slave proces pouze přijímá pomocí metody *scatter* subpopulaci částic, pro něž vypočte chyby a tyto chyby pošle zpět pomocí metody *gather*.

```
Inputs:netParameters, algorithm parameters
Outputs:newNetParameters

public double[] ParallelPso(double[] netParameters)
{
    set the first position to netParameters other randomize
    randomize velocities of all particles
    for(int it = 0; it < maxIterations; it++)
        Create subpopulations of particles positions
        Send with scatter to all processes include master
        Calculate error of own part of subpopulation
        Gather errors of all particles
        Set pBest and gBest
        c0 = cStart − ((cStart − cEnd) ∗ it)
        for (int i = 0; i < particlesCount; i++)
            for (int j = 0; j < dimension; j++)
                r[i][j] = c0 ∗ r[i][j] + c1 ∗ rand ∗ (pBest[i][j] − s[i][j])
                                       + c2 ∗ rand ∗ (gBest[j] − s[i][j])
                s[i][j] = s[i][j] + r[i][j]
    return newNetParameters
}
```

Výpis 2: Pseudokód paralelní verze PSO algoritmu

2.4.2 Simulované žíhání (Simulated annealing)

Simulované žíhání (SA) je další algoritmus pro nalezení globálních extrémů. Tentokrát se nejedná o algoritmus založený na populaci jedinců. Je zde pouze jedno řešení které se pohybuje v prostoru všech možných řešení. Tento algoritmus je inspirován žíháním v metalurgii, kdy se pomalým ochlazováním žhavého kovu dospěje ke stabilnější krystalové mřížce. Toto umožňuje dospět k lepším vlastnostem kovů.

Algoritmus pro každou teplotu, která se časem snižuje, provádí několik kroků Metropolisova algoritmu. Ten funguje tak, že pokud se vybere soused od daného řešení, který má lepší hodnotu zdatnosti, tak je automatiky přijat. Pokud ale má horší zdatnost, tak je přijat jen s pravděpodobností viz rovnice 16. Díky tomu je při vysokých teplotách možné vyskočit z lokálního extrému a s ochlazováním teploty se tato pravděpodobnost snižuje a jsou přijímány častěji jen řešení s lepší hodností. Existuje i upravená verze tohoto algoritmu kdy se pomatuje dosavad nejlepší řešení z celého běhu simulovaného žíhání,

protože při snížení teploty k teplotě blízké 0, je šance přijetí i horšího řešení, ikdyž je tato pravděpodobnost malá [9, 13].

$$P(\vec{s} \to \vec{s_0}) = \begin{cases} 1 & \text{if } f(\vec{s}) < f(\vec{s_0}) \\ e^{-(f(\vec{s})-f(\vec{s_0}))/temp} & \text{if } f(\vec{s}) \geq f(\vec{s_0}) \end{cases} \tag{16}$$

kde
$P(\vec{s} \to \vec{s_0})$ ... pravděpodobnost přijetí lepší pozice

V publikaci *Parallelizing simulated annealing algorithms based on high-performance computer* [14] autoři porovnávali několik paralelních přístupů k simulovanému žíhání. Zkoušeli třeba paralelizovat pohyb, kdy se vydal algoritmus z jednoho startovního místa do několika jiných míst zároveň, nebo z několika startovních míst paralelně do dalších, ale tato řešení nebyla vhodná pro velké problémy. Proto navrhli kombinaci GA a SA, který z jejich výsledků byl efektivnější.

Tento algoritmus funguje tak, že se vygeneruje počáteční populace a nechá se nad ní provádět GA algoritmus po několik generací na procesu s rankem 0. Výsledná generace se rozešle na ostatní procesy. Na jednotlivých procesech se začne provádět algoritmus SA, poté se výsledky pošlou zpět na rank 0, který provede GA algoritmus nad nimi, ale tentokrát jen nejlepší řešení, pokud nespluňuje koncové podmínky, je rozesláno opět ostatním procesům, kde se provede SA.

2.4.3 Genetický algoritmus (Genetic algorithm)

Genetický algoritmus (GA) je algoritmus inspirovaný přírodou a to evolucí popsanou Charlesem Darwinem. Zakladatelem algoritmu je John Holland [1, 9]. Algoritmus pracuje s populací jedinců. Každý jedinec je reprezentován řetězcem (chromozomem), který obsahuje parametry (geny) řešení daného problému. Geny mohou nabývat různých datových typů, často binární nebo reálné. Populace se vyvíjí časem a to tak, že do nové generace jsou tvořeni noví jedinci podle tří základních operací:

- **Křížení**-Je základní operace, při které se vyberou typicky dva rodiče. Kombinací jejich chromozomů jsou vytvořeni dva potomci. Je mnoho možností jak chromozomy zkombinovat. Jeden způsob je že se vygeneruje náhodně index, který chromozom rodíčů rozdělí na dvě části a poté je levá čast chromozomu otce spojena s pravou částí chromozomu matky. Tímto je vytvořen chromozom prvního potomka. Analogicky je ze zbylých částí vytvořen druhý potomek.
- **Mutace**-Po vytvoření potomka je šance, že se provede nad jeho chromozomem náhodná změna genu. Mutace umožňuje díky těmto změnám nalézt i jiná řešení, než je možné pouhým křížením. Umožňuje se dostat z lokálního minima.
- **Reprodukce**-Pří reprodukci je vybraný jedinec přesunut do nové generace.

Výběr jedinců ke křížení se provádí pomocí různých náhodných metod, ale které nějak upřednostňují silnější jedince. Těchto metod je více. Základní je ruletový systém (Rulete Wheel Selection), ten má však nevýhodu, že příliš upřednostňuje nejsilnějšího jedince a ostatní nemají téměř šanci být vybráni a to není dobré pro zachování pestrosti populace.

Jiný systém nazývající se pořadový (Rank Selection) je vhodnější, populace je seřazena od nejsilnějšího jedince a každý dostane rank nejslabší 1, další 2 atd. Poté jsou sečteny všechny ranky do proměnné *suma*. Dále je vygenerováno náhodné číslo *r* v intervalu (0, *suma*). Poté je procházena populace a sčítají se tyto ranky do nové proměnné *actual*, pokud tato hodnota překročí číslo v *r*, je tento jedinec vybrán.

Modifikací tohoto algoritmu může být Elitismus. To znamená, že několik nejsilnějších jedinců je automaticky zkopírováno do nové populace. Tento způsob umožňuje uchovat nejlepší řešení, jinak není zaručeno, že nejsilnější jedinec bude v nové generaci nebo jeho potomci. Tento způsob i zrychluje konvergenci algoritmu.

V neuronových sítí je potřeba kódovat váhy a parametry aktivačních funkcí do chromozomu. Důležité parametry tohoto algoritmu jsou velikost populace, maximální počet generací a parametry na ovlivnění šance na křížení, mutaci a reprodukci.

2.4.4 DE

Diferenciální evoluce je algoritmus podobný genetickému algoritmu. Zakladateli tohoto algoritmu jsou Ken Price a Rainer Storm. Je zde opět populace jedinců, která představuje možná řešení daného problému. Narozdíl od GA se mutace a křižení provádí dohromady. K vytvoření nového jedince je potřeba čtyř rodičů a ne jen dvou [9]. Tento algoritmus byl porovnán ve studii autorů Abdual-Salam, Abdul-Kader a Abdel-Wahed s algoritmem PSO pro učení neuronové sítě na problému předpověď i časové řady, kde DE algoritmus podával lepší výsledky [16].

Algoritmus funguje tak, že se vygeneruje počateční populace náhodně v prostoru možných řešení. Pro neuronové sítě opět jedinec představuje pole reálných čísel které představují hodnoty vah a parametrů aktivačních funkcí. Pro každého jedince počátční populace je vypočtena zdatnost. Je několik variant diferenciálních evolucí. V této práci používám typ označován jako *DE/best/1/bin*. Toto označuje tvorbu šumového vektoru. V každé generaci probíhá cyklus procházející jedince. Jedinec je označen jako aktivní a hraje roli při tvorbě nového jedince. K tomuto jedinci jsou vybráni další tři jedinci jeden je nejlepši jedinec, ostatní jsou vybráni náhodně, musí však být všichni čtyři vzájemně různí. Z těchto tří vybraných jedinců se vytvoří šumový vektor podle rovnice 17.

$$ns_j = s_{best,j} + F \cdot (s_{r_1,j} - s_{r_2,j}) \tag{17}$$

Nový jedinec se poté vytvoří z aktivního jedince a šumového vektoru tak, že se prochází prvky vektoru a vygeneruje se náhodné číslo od 0 do 1, pokud je toto číslo menší

než konstanta CR (práh křížení) tak nový jedinec bude obsahovat parametr z šumového vektoru, jinak bude obsahovat parametr z aktivního jedinec. Nový jedinec musí ale obsahovat minimálně jeden parametr z šumového vektoru, jinak by neměl smysl nasledný krok. Nový jedicen (nazýván zkušebním vektorem) musí být poté podroben ještě zkouškou. Pokud má lepší hodnotu zdatnosti než aktivní jedinec, je přesunut do nové generace, v opačném případě se přesouvá do nové generace aktivní jedinec.

Algoritmus je ovlivněn parametry velikosti populace, mutační konstantou F, prahem křížení CR a počtem generací. velikost populace a počet generací je nutné zvolit podle problému. Pro paralelní verzi algoritmu by měla být populace volena jako násobek počtu procesů. Mutační konstanta se volí v rozmezí intervalu (0,1). Podle nových zkušeností je dobré volit F náhodně v intervalu (0.5, 1) pro každou generaci. Konstanta CR ovlivňuje kolik informace bude obsahovat zkušební vektor z aktivního jedince a kolik z šumového vektoru. Větší hodnota CR (0.5,1) znamená, že více informace půjde z šumového vektoru.

Při hledání řešení např. parametrů pro neuronovou síť, nemusí být jen podmínka ukončení algoritmu dosažení maximálního nadefinovaného počtu generací, ale může to být dosažení akceptovatelné zdatnosti. Často se populace dostane do minima, ať už lokálního nebo globálního v tom případě nemá smysl pokračovat v algoritmu. Stačí k tomu kontrola v každé generaci pokud se bude shodovat zdatnost nejlepšího a nejhoršího jedince.

Paralelní verze algoritmu funguje podobně s rozdílem, že se nechají vytvořit zkušební vektory a k těmto vektorům je vypočtena paralelně zdatnost, tím že se zkušební vektory rovnoměrně rozešlou metodou *Scatter* v MPI na všechny procesy. U flexiblilních neuronových stromů se rozešle na všechny procesy ze začatku běhu algoritmu jen jednou DNA jedince, který je učen, poté při běhu se zasílají pouze parametry. Na všech procesech se poté spustí výpočet chyby všech neuronových stromů jako MSE nebo jiné. Metodou *Gather* jsou poté navráceny chyby, které jsou použity jako zdatnost. Poté pokračuje standartně algoritmus tvorbou nové generace. Pseudokód diferenciální evoluce je možné vidět ve výpisu 3.

```
Inputs:netParameters, algorithm parameters
Outputs:newNetParameters

public double[] De(double[] netParameters)
{
    set the first netParameters to first individual other randomize
    evaluate population
    for(int it = 0; it < maxGeneration; it++)
        find best
        for(int j = 0; j < populationSize; j++)
            select 3 different parents, first is best
            create noisy vector from parents
            create trial vector and evaluate
            if ( trialVecotor .Fitness < individual [j ])
                replace individual [j ] with trial vector
        if (best.Fitness < required)
            break
    return best.Parameters
}
```

Výpis 3: Pseudokód DE algoritmu

2.5 Použití

FNT model je vhodný na problémy, jako jsou předpovědi časových řad, klasifikační problémy, aproximace funkcí, ... Tento model dokáže v těchto problémech dosahovat vyšší přesnosti i zobecnění díky tomu, že dokáže identifikovat důležité vstupy a nedůležité zanedbat.

Na problém klasifikace byl FNT použit k detekci rakoviny ve studii nazvané *Multiclass classification of microarray data samples with Flexible Neural Tree* [4]. Zde autoři Xuejiao Lei a Yuehui Chen testovali algoritmus FNT, pro optimalizaci struktury byl použit algoritmus PIPE a na optimalizaci parametrů algoritmus PSO. Testování bylo provedeno na dvou trénovacích množinách, první řešila klasifikaci tří druhů leukémie a druhá řešila klasifikaci tří druhů rakoviny lymfatického systému. Oproti jiným modelům měl jejich model nejlepší přesnost, především pro detekci leukémie dosahoval FNT model 100%.

K předpovědi časové řady byl FNT model ve studii *Tree-Structure based Hybrid Computational Intelligence: Theoretical Foundations and Applications* [1] použit například na předpověď koncentrace oxidu uhličitého obsaženého v plynu vycházejícího z plynového kotle v závislosti na průtoku plynu vstupujícího do kotle (Jenkins-Box řada J). Tato data se vyskytují v mnoha studijích a slouží tedy dobře k porovnání různých metod.

3 HPC

Klasické počítače mají na některé úlohy nedostatečný výkon díky výkonu procesoru, který není zvyšován dostatečně rychle, proto je jednodušší vyvíjet procesory o více jádrech. Využití výkonu vícejádrových procesorů jedním procesem je možné použitím vláken v programu. Ale i tak tento výkon není dost velký, proto v HPC je propojeno velké množství takovýchto počítačů (uzlů), k dosažení mnohem většího výkonu. Vede to ale k problému jak paralelizovat algoritmus, aby mohl využít výkon takového superpočítače. Proto byl vyvinut standart MPI, kdy je spuštěno na jednotlivých uzlech nebo i jádreh mnoho procesů, které mezi sebou moou komunikovat zasíláním zpráv.

3.1 MPI

MPI je standard, jedná se o specifikaci, jak má vypadat knihovna pro paralelizaci algoritmů pomocí komunikace (zasílání zpráv) mezi procesy. Oproti vláknům zde není sdílena paměť, ale jsou zde většinou stejné procesy, každý pracující nad různými daty (SPMD paralelní model). Implementací tohoto standardu jsou například Open MPI nebo MS-MPI, jako programovací jazyky se používají C, C++ nebo Fortran [18].

3.2 MPI.NET

Jedná se o .NET knihovnu umožňující používat MPI komunikaci k tvorbě paralelních aplikací spustitelných na Windows klastrech. Tuto knihovnu je možno používat v jazycích .NET jako jsou C#, Visual Basic, ... Tato knihovna zaobaluje MS-MPI, která je implementací MPI standardu společnosti Microsoft, s tím že metody této knihovny se snaží dodržovat jednoduchost a konvence jazyka C#. Tato knihovna je nicméně použitelná i na systémech s jinými imlementacemi MPI, jako jsou Open-MPI nebo MPICH2 [17], [12].

Na Windows systémech je potřeba mít kromě MPI.NET knihovny nainstalovanou MPI implementaci MS-MPI, tu je možné nainstalovat s balíkem HPC Pack 2008 nebo 2012. Vytvořený program je poté možné spustit vícekrát příkazem *mpiexec*. Každému procesu je přiřazen "rank", což je *Integer* hodnota identifikující proces. První proces ma hodnotu začínající nulou. První proces se volí většinou jako Master proces, pokud je použit Master-Slave model, tak Master proces provádí základní koordinující činost a výpočetně náročná část algoritmu se provádí paralelně na ostatních Slave procesech, kde každý proces zpracovává část dat.

Program používající MPI komunikaci musí mít určitou strukturu. Pokud je použit jazyk C# a vývojové prostředí Visual Studio, tak prvním krokem je vytvoření konzolové aplikace. Po vytvoření tohoto projektu je potřeba přidat referenci na knihovnu MPI.NET a zahrnout ji v programu. Struktura Master-Slave programu potom vypadá většinou jak je možné vidět ve výpisu 4. Každý proces musí inicializovat prostředí MPI, aby mohl používat MPI komunikaci, to je provedeno vytvořením objektu *Environment* a předáním mu

argumentů z konzole, při vytváření objektu je použito bloku *using*, aby došlo ke správnému uvolnění z paměti před ukončením programu. V tomto bloku je získán objekt *Intracommunicator*, který obsahuje metody pro komunikaci mezi všemi vytvořenými procesy. Komunikátor znamená skupina procesů, které mohou mezi sebou komunikovat. V MPI může být vytvořeno více komunikátorů (skupin procesů). Objekt *Intracommunicator* tedy slouží k posílání zpráv mezi procesy v dané skupině. Tento objekt má vlastnost *Rank* k identifikaci procesu. Ke komunikaci tento objekt obsahuje několik metod, dále popíšu jen nejdůležitější z nich.

```
using System;
using MPI;
class Program
{
    static void Main(string[] args)
    {
        using (new MPI.Environment(ref args))
        {
            Intracommunicator comm = Communicator.world;
            if (comm.Rank == 0)
            {
                //Master process code
            }
            else
            {
                //Slave process code
            }
        }
    }
}
```

Výpis 4: Struktutra Master-Slave programu používající knihovnu MPI.NET

Metoda *Send* slouží k poslání zprávy nějakému procesu. Tato metoda je blokující, tedy dokud není zpráva přijata, tak nemůže být proveden příkaz za jejím voláním. Hlavičku metody je možné vidět ve výpisu 5. Jedná se o generickou metodu, je možné s ní posílat primitivní datové typy ale i objekty. Objekty se v této metodě serializují a posílají jako pole *byte*. Proto je nutné třídu označit jako *Serializable*. Parametr *value* je tedy hodnota, kterou chceme poslat, parametrem *dest* se označuje proces, kterému chceme zprávu poslat (Rank toho procesu). Parametr *tag* může být použit k určité filtraci zpráv, pokud je nastaven na hodnotu 0, tak tuto zprávu dokáže přijmout pouze *Receive* metoda se stejnou hodnotou *tag*. S využitím polymorfismu se v této knihovně nachází více druhů této metody. Pro bližší informace je ale lepší si projít dokumentaci na stránkách univerzity v Indianě [17].

```
public void Send<T>(
    T value,
    int dest,
    int tag
)
```

Výpis 5: Metoda Send knihovny MPI.NET

Metoda *Receive* tedy slouží k přijímání zpráv odeslaných metodou *Send*. Hlavičku metody je možné vidět ve výpisu 6. Jedná se opět o generickou metodu, kterou je možné přijímat zprávy různých promitivních datových typů i celé objekty. Tato metoda vrací přijatý objekt a parametry má zdroj *source*, ten znamená z jakého procesu je očekávo přijetí zprávy a parametr *tag* byl popsán u metody *Send*. Jedná se o blokující metodu, tedy dokud není přijata nějaká zpráva, nemůžou být prováděny příkazy za jejím voláním.

```
public T Receive<T>(
    int source,
    int tag
)
```

Výpis 6: Metoda Receive knihovny MPI.NET

Metoda *Broadcast* slouží k poslání zprávy z nějakého procesu na všechny další procesy v daném komunikátoru, hlavička metody je ve výpisu 7. Jedná se o generický typ metody. Parametr *root* je nastaven na hodnotu *Rank* procesu, který danou zprávu odesílá. Ostatní procesy tímto voláním metody zprávu přijmou. U odesílatele je zpráva přečtena v této metodě z proměnné *value*. U ostatních procesů je do parametru *value* zapsána přijatá hodnota.

```
public void Broadcast<T>(
    ref T value,
    int root
)
```

Výpis 7: Metoda Broadcast knihovny MPI.NET

Metoda *Scatter* má dvě verze, jedna slouží k odesílání na Master (Root) procesu viz výpis 8, druhá verze slouží k přijímání na ostatních procesech viz výpis 9. Tato metoda slouží k rozeslání pole na ostatní procesy tak, že *i*-tý prvek pole je poslán na *i*-tý proces. Tato metoda slouží k rozdělení pole pro jednotlivé procesy, kde může na základě těchto hodnot být proveden výpočet. První verze pro Master proces v parametru odešle pole hodnot a první hodnotu zároveň vrátí Master procesu, aby se také účastnil výpočtu. Ostatní procesy použijí druhou metodu, kde pouze specifikují *Rank* Master procesu a přijmou danou část pole.

```
public T Scatter<T>(
    T[] values
)
```

Výpis 8: Metoda Scatter pro Master proces knihovny MPI.NET

```
public T Scatter<T>(
    int root
)
```

Výpis 9: Metoda Scatter pro Slave procesy knihovny MPI.NET

S předešlým typem metody se váže metoda *Gather*, ta slouží k sesbírání jednotlivých hodnot na všech procesech do pole na Master procesu. Tato metoda je generická, může

tedy posílat i serializovatelné objekty. Hlavička metody je ve výpisu 10. Parametr *root* určuje rank Master procesu a *value* hodnotu, kterou chceme poslat. Tato metoda vrací pole sesbíraných hodnot na Master procesu, na ostatních vrací hodnotu *null*.

```
public T[] Gather<T>(
    T value,
    int root
)
```

Výpis 10: Metoda Scatter pro Slave procesy knihovny MPI.NET

Další často používanou metodou je metoda *Barrier*, tato metoda nic nevrací a nemá ani žádný parametr. Slouží ale k tomu aby došlo k sesynchronizování procesů. Pokud je tato metoda zavolána někde v kódu na Master procesu, tak nejsou prováděny následující příkazy dokud není tato metoda zavolána i na všech ostatních procesech.

Metoda *ImmediateSend* viz výpis 11 slouží k asynchronnímu poslání zprávy. To znamená, že po jejím zavolání není potřeba čekat na přijetí této zprávy cílovým procesem, ale je mezitím možné dělat jinou práci. Tato metoda vrací objekt *Request*, přes který je možné počkat na doručení zprávy. Tyto objekty třídy *Request* je vhodné vkládat do kolekce *RequestList*, na kterou je možné zavolat metodu *WaitAll*, ta počká dokud nejsou všechny tyto asynchronní zprávy doručeny. Před ukončením algoritmu by se mělo počkat, než bude všechna komunikace dokončena, jinak je vyvolána výjimka.

```
public Request ImmediateSend<T>(
    T value,
    int dest,
    int tag
)
```

Výpis 11: Metoda ImmediateSend knihovny MPI.NET

Metoda *ImmediateReceive* viz výpis 12, poté slouží k přijímaní zpráv poslaných metodou *ImmediateSend*. Tato metoda vrací objekt *ReceiveRequest*, který čeká na doručení zprávy, mezitím je možné opět řešit jinou práci. Pokud chceme otestovat, zda zpráva byla přijata je možné na tento vrácený objekt zavolat metodu *Test*. Metoda *Test* vrací hodnotu *null* pokud nebyla ještě přijata žádná zpráva, pokud byla přijata, tak vrací objekt třídy *CompletedStatus*. Poté je možné pomocí objektu *ReceiveRequest* získat zprávu zavoláním metody *GetValue*. Pokud potřebujeme zjistit zdroj této zprávy, je nutné použít vrácený objekt *CompletedStatus* a jeho vlastnost *Source*.

```
public ReceiveRequest ImmediateReceive<T>(
    int source,
    int tag
)
```

Výpis 12: Metoda ImmediateReceive knihovny MPI.NET

4 Implementace knihovny

Tato část práce se bude zabývat tvorbou knihovny. Je rozdělen na několik částí. První pojednává o požadavcích kladených na knihovnu. Druhá se zabýva analýzkou požadavků, která bude použita a rozpracována podrobněji dále v návrhu. Poslední částí bude implementace, ve ktreré budou popsány detaily důležitých tříd knihovny.

4.1 Požadavky

Cílem této práce je vytvořit knihovnu, která bude sloužit k nalezení jednoho nebo více neuronovéch stromů pro problém ve formě trénovací množiny, kterou uživatel zadá této knihovně. Protože nalezení řešení problému je výpočetně složité, bude knihovna paralelizovaná pomocí MPI. Nalezené řešení bude vráceno uživateli knihovnou. Uživatel bude moci ovlivňovat řídící parametry algoritmů pro optimalizaci struktury a optimalizaci parametrů a parametry flexibilních neuronových stromů. Nezáleží na volbě programovacího jazyka, ale bylo by vhodné knihovnu otestovat na superpočítači ANSELM.

4.2 Analýza požadavků

Pro optimalizaci struktury flexibilních neuronových stromů bude knihovna používat algoritmus Genetické programování viz sekce 2.3.1. K optimalizaci parametrů bude použit algoritmus Rojení částic viz sekce 2.4.1, knihovna však bude moci být rozšířena i o jiné algoritmy pro možnost porovnání.

Optmalizační algoritmy budou paralelizované pomocí MPI. Protože jsou to algoritmy s populací jedinců, je možné nejnáročnější část algoritmu výpočet chyby sítě provádět paralelně. Paralelizace bude využívat techniku Master-Slave, kdy proces s rankem 0 bude Master a bude provádět samotný algoritmus optimalizace struktury a parametrů, ostatní procesy budou Slave a budou sloužit k výpočtu chyby sítě pro část populace, část populace k výpočtu chyby zůstane i na Master procesu, aby nebyl nevyužitý.

Knihovna bude informovat o aktualní chybě nejlepšího stromu uživatele pomocí událostí. Dále bude informovat v jaké generaci je algoritmus Genetického programování nebo v jaké iteraci je algoritmus Rojení částic.

Nalezené řešení bude tvořit objekt neuronové sítě, ten bude obsahovat jeden nebo více objektů neuronových stromů, podle toho kolik má trénovací množina výstupních parametrů. Tento objekt neuronové sítě bude knihovna vracet po dokončení hledání. Pokud knihovna nestihne nalézt neuronový strom s požadovanou chybou

Knihovna dále bude implementovat několik druhů výpočtu chyby sítě, jako jsou MSE a RMSE popsané dříve. Dále bude implementovat více druhů aktivačních funkcí, minimálně však Hyperbolický tangens pro použití sítě na výstupy v intervalu (-1,1) a Sigmodální funkci pro výstupy v intervalu (0,1).

Uživatel bude moci nastavit maximální povolenou hloubku stromu, který může vzniknout při hledání pro omezení prostoru hledaných řešení, dále nastavovat parametry optimalizačních algoritmů (PSO, GP...), akceptovatelnou chybu neuronového stromu a počet epoch FNT algoritmu. U GP algoritmu by měla být možnost nastavit parametr penalizace, který bude upravovat zdatnost neuronového stromu tak, že pokud mají dva stromy stejnou chybu, tak menší z nich bude mít lepší zdatnost.

4.3 Návrh knihovny

Knihovna bude programována v programovacím jazyce C# s využitím knihovny MPI.NET pro podpou MPI. v prostředí Visual Studio 2013. Kód bude programován tak, aby jej bylo možné spustit na superpočítači ANSELM za pomoci nainstalované verze MONO, protože na něm běží linuxová distribuce BullX.

Základní jmenný prostor knihovny bude *MpiFntLibrary*. V něm bude základní třída pro využívání knihovny *Fnt*, kterou bude moci uživatel najít řešení na zadaný problém.

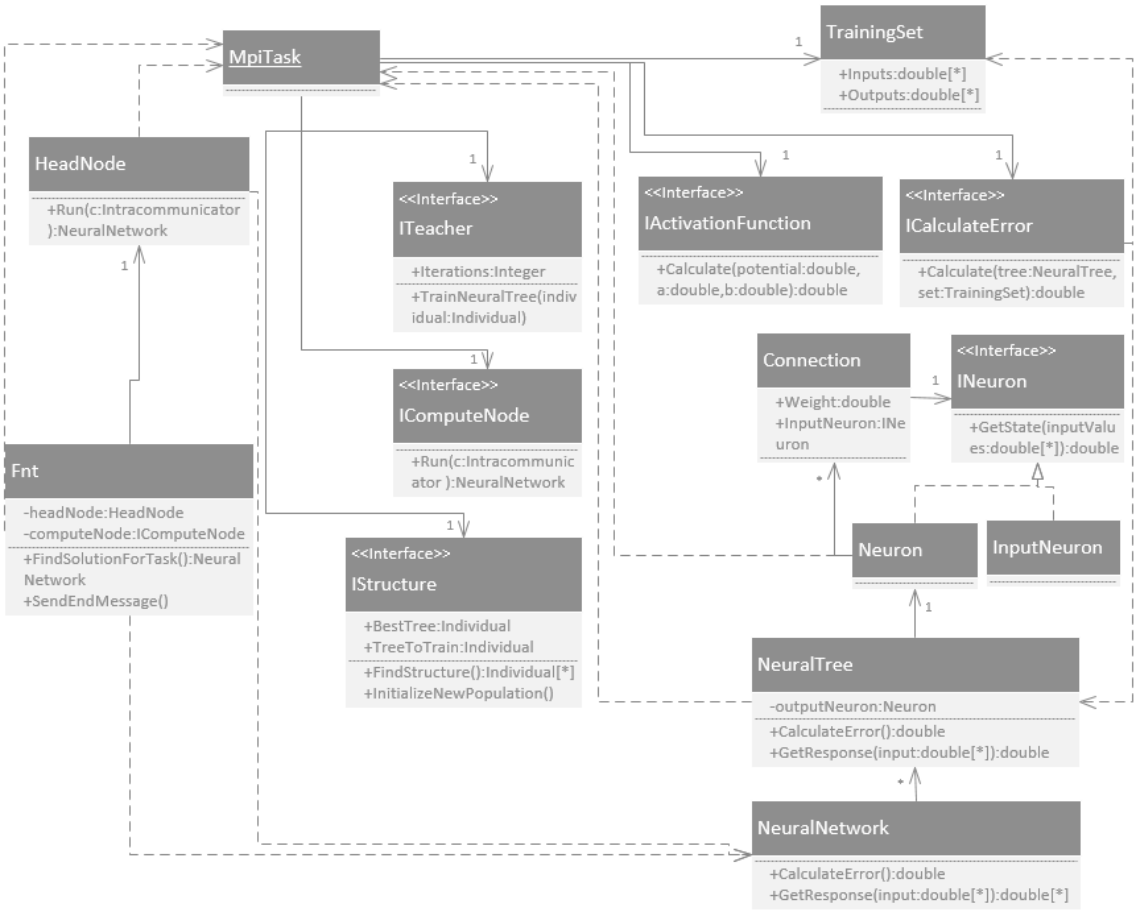
V základním jmenném prostoru budou další jmenné prostory podle funkčnosti. Tyto jmenné prostory budou *ActivationFunctions*, *Data*, *Exceptions*, *Global*, *Learning*, *Mpi*, *Network*, *Utilities*.

Jmenný prostor *ActivationFunctions* bude obsahovat různé aktivační funkce, které bude možné rozšířit uživatelem o vlastní, implementují rozhraní *IActivationFunction*. Jm. prostor *Data* bude obsahovat třídy pro trénovací množinu a aktualně zpracovávaný vzor trénovací množiny. Jmenný prostor *Exceptions* bude definovat výjimky knihovny.

Další jm. prostor *Global* bude obsahovat globalní statické třídy *Info* pro informace o průběhu hledání řešení jako jsou čas, počet vytvořených neuronových stromů atp., *RandomGenerator* pro generování náhodných čísel a hlavně třída *MpiTask*, která bude sloužit k definování problému a k výběru optimalizačních algoritmů ať už z knihovny nebo jiné implementované uživatelem.

Důležitý jmenný prostor bude *Learning*, ten bude obsahovat jmenné prostory *ParameterOptimization* pro algoritmy optimalizující parametry neuronového stromu a *StructureOptimization* pro hledání vhodné struktury neuronového stromu.

Jmenný prostor *MPI* bude obsahovat třídu *HeadNode* provádějící metody na Master procesu a *ComputeNode* ta bude vykonávat metody na Slave procesech.



Obrázek 4: Třídní diagram knihovny

Další jm. prostor *Utilities* bude obsahovat rozhraní *ICalculateError* definující jak mají vypadat třídy pro výpočet chyby sítě, implementovat toto rozhraní budou třídy *MeanSquare* a *RootMeanSquare*, další třídy bude moci implementovat uživatel, pokud mu nebudou tyto základní dostačovat.

Poslední jmenný prostor bude *Network* obsahující třídy pro definování a používání neuronových stromů. Ten bude obsahovat třídy *NeuralNetwork* složená s tříd *NeuralTree*. Neuronový strom bude složen s neuronů implementující *INeuron* rozhraní. Neurony Budou dvou typů *InputNeuron* a *Neuron*. Mezi neurony bude vazba spojení třídou *Connection*, která se bude nacházet také v tomto jm. prostoru.

Na obrázku 4 je možné vidět návrh knihovny pomocí třídního diagramu, jsou zde hlavní třídy a metody, ostatní byly vypuštěny z důvodu přehlednosti.

4.4 Implementace

V této kapitole se budu zabývat implementací knihovny. Popíšu zde detaily jednotlivých tříd. Třídy budou seřazeny podle jmenných prostorů, kde se nacházejí.

4.4.1 Třídy *Sigmoid*, *HyperbolicTangent*, *Gaussian*

Tyto třídy slouží k výpočtu aktivační funkce neuronu. Všechny musí implementovat rozhraní *IActivationFunction*, aby mohly být použity ve třídě *MpiTask*. Rozhraní *IActivationFunction* definuje dvě metody *Calculate*, obě vrací číslo typu *double*, liší se ale v parametrech, jedna metoda má jeden parametr typu *double*, který znamená potenciál neuronu, druhá metoda má kromě tohoto parametru další dva, které jsou parametry aktivační funkce *a* a *b* taky typu *double*. Třídy podle názvu znamenají aktivační funkce Sigmoidální, Gaussovu a funkci Hyperbolický tangens popsaných v dříve.

4.4.2 Třída *TrainingSet*

Třída *TrainingSet* obsahuje trénovací množinu, má dvě vlastnosti *Outputs* a *Inputs*, které představují dvourozměrné pole. Protože neuronový strom má jeden výstupní neuron, je potřeba při učení vědět, který výstup z trénovací množiny mu náleží, to udává statická třída *MpiTask*, ta obsahuje proměnnou *OutputIndex* typu *Integer* udávající který výstup je zpracováván.

4.4.3 Třídy *Info*, *MpiTask*, *RandomGenerator*

Statická třída *RandomGenerator* slouží ke generování čísel z důvodu, že náhodná čísla je nutné využívat v různých částech kódu a kdyby se nepoužila jedna instance třídy *Random* v této třídě, docházelo by ke generování stejných čísel. Tato třída ma metodu *GetRandomNumber*, ta má 2 paramtry typu *double min* a *max*, vracející číslo typu *double* v intervalu $[min, max)$. Třída *Info* slouží k shromáždění informací o průběhu hledání řešení FNT algoritmu, obsahuje vlastnosti jako jsou celková doba optimalizace parametrů *LearningTime* a struktur *StructureTime*, vlastnost *StructureTreesCreated* udává kolik bylo vytvořeno nových neuronových stromů při Genetickém programování u křížení. Velice důležitá třida z pohledu uživatele je statická třída *MpiTask*. Díky této třídě uživatel definuje použitý algoritmus pro optimalizaci parametrů vlastností *Teacher* typu *ITeacher*, aktivační funkci použitou neurony přes vlastnost *ActivationFunction* typu *IActivationFunction* a typ výpočtu chyby neuronové sítě přes vlastnost *ErrorType* typu *ICalculateError*. Definuje se zde i maximální počet epoch FNT algoritmu, minimální požadovaná chyba a počet kroků algoritmu učení na doučení nalezeného řešení. Dále je zde možné nastavit inicializační hodnoty parametrů sítě, jako jsou váhy a parametry aktivačních funkcí, pro ně se volí hodnoty většinou z intervalu $(-1, 1)$. Dále je zde možné omezit minimální a maximální hodnotu těchto parametrů.

4.4.4 Třídy *MpiDifferentialEvolution*, *MpiParticleSwarm* a rozhraní *ITeacher*

Rozhraní *ITeacher* definuje jak má vypadat třída optimalizující parametry neuronového stromu. Definuje vlastnost *Iterations* udávající počet iterací algoritmu učení, pro diferenciální evoluci nebo geneticý algoritmus tento parametr znamená počet generací.

Třída *MpiParticleSwarm* implementuje algoritmus popsaný v kapitole 2.4.1 a to jeho paralelní verzi. Tento algoritmus implementuje několik veřejných vlastností pro nastavení algoritmu:

- **WStart, WEnd** - slouží jako okrajové podmínky pro parametr setrvačnosti $c0$, který je každou iteraci lineárně snižován na hodnotu *WEnd*. Setrvačnost zpomaluje aktuální rychlost částice. Datový typ vlastností je *Double*.
- **C1, C1** - tyto vlastnosti určují kam má částice častěji směřovat. Pokud je vyšší hodnota vlastnosti *C1*, tak částice častěji směřuje ke svoji nejlepší nalezené pozici, jinak ke globální nejlepší pozici. Hodnoty vlastností jsou voleny v základu na hodnotu 2.0. Datový typ těchto vlastností je *Double*.
- **MaxVelocity** - udává maximální i minimální hodnotu rychlosti částice. Datový typ vlastnosti je *Double*.
- **RangeMin, RangeMax** - tyto vlastnosti udávají minimální a maximální hodnotu pozice částice, omezují hodnoty prvků polí pozic částic. Jejich datový typ je *Double*.
- **Iterations** - tato vlastnost udává počet iterací algoritmu Rojení Částic (PSO). Datový typ vlastnosti je *Integer*.

Základní metodou je metoda *TrainNeuralTree*, která má dva vstupní parametry, jeden je typu *Individual* a druhý je typu *Intracommunicator*. Objekt *Individual* této metodě předává jedince, kterého je potřeba naučit, obsahuje dvě vlastnosti *Dna* typu *String* a *Error* typu *Double*, z jeho DNA je možné získat parametry a strukturu neuronového stromu. Objekt *Intracommunicator* je potřebný z důvodu MPI komunikace, kvůli posílání, přijímaní zpráv a jiných možností prostředí MPI popsaných v kapitole 3.2. Parametry je možné získat z jedince tak, že se nechá vytvořit objekt *NeuralTree* pojmenovaný jao *workTree* statickou metodou této třídy *CreateTree* s parametrem typu *String*, který znamená DNA řetězec a poté je na tento objekt zavolána metody *GetParameters* pro získání parametrů ve formě pole *Double* hodnot.

Potom je zavolána metoda *InitializeComputeNodes* tu popíši později, protože je stejná jako u agoritmu u třídy *MpiDifferentialEvolution*. Hned po ní je zavolána metoda *Initialize*, která má parametr přebírající parametry jedince. Všechny pole v této metodě jsou typu *Double*. V této metodě se inicializují dvourozměrné pole *particlesPositions*, *particlesVelocities*, *particlesBestPositions*, první rozměr má velikost počtu částic definovaný proměnnou *particlesCount*, druhý rozměr má velikost podle dimenze problému v proměnné *dimension*, jedná se o počet parametrů neuronového stromu. Pole *particlesPositions* ucho-

vává pozice částic, pole *particlesVelocities* uchovává rychlosti částic a pole *particlesBestPositions* uchovává dosavadní nejlepší pozici pro každou částici. Dále jsou inicializována jednorozměrná pole typu *Double*, jako jsou *particlesErrors* uchovávající chyby každé částice, *particlesBestErrors* uchovávající dosavad nejnižší chyby každé částice, tato pole mají velikost podle počtu částic. Dále je zde pole *globalBestPosition*, které znamená dosavadní nejlepší pozici (s nejnižší chybou) částice. Proměnná *globalBestError* typu *Double* je nastavena na maximální hodnotu *Double.MaxValue*. Parametry jedince přijatých touto metodou jsou nastaveny na první index polí *particlesPositions* a *particlesBestPositions*. Parametry jsou jinak u těchto polích nastaveny náhodně v rozmezí ohraničením vlastnostmi *RangeMin* a *RangeMax*, chyby jsou nastaveny na hodnoty *Double.MaxValue*.

Dále následuje hlavní cyklus PSO algoritmu s maximem iterací definovaných parametrem *maxIterations*. V tomto cyklu se přepočítává setrvačnost rychlosti podle rovnice 15. Po tomto přepočtu je volána metoda *Evaluate* ta se shoduje s metodou v třídě *MpiDifferentialEvolution*, proto ji popíšu později, slouží ale k rozeslání populace částic na všechny uzly a tam jsou vypočteny chyby. Poté je zavolána metoda *CheckParticlesErrors*. V této metodě probíhá cyklus procházející všechny částice a v něm jsou podmínky kontolující zda chyba částice (v poli *particlesErrors*) je menší jak dosavadní nejlepší chyba dané částice (v poli *particlesBestErrors*) a poté druhá podmínka kontrolující , jestli chyba částice (v poli *particlesErrors*) je nížší než globální chyba (v proměnné *globalBestError*), pokud je podmínka splněna je nastavena lepší hodnota chyby. Poté je spuštěn vnitřní cyklus procházející všechny částice a v něm další vnitřní cylkus procházející dimenze. V tomto vnitřním cyklu jsou pro každou částici přepočteny pole rychlostí a pozic podle rovnic 13 a 14 po každém přepočtu je zkontrolováno zda se nachází rychlost v rozmezí $[-maxVelocity, maxVelocity]$ pokud je větší jak tato hranice je nastavena rychlost na hodnotu *maxVelocity*, pokud je menší tak na hodnotu *-maxVelocity*. Pozice je kontrolována, zda je v rozmezí $[-rangeMin, rangeMax]$, pokud je mimo tuto hranici, tak je zvolena pozice náhodně v této hranici. Na konci iteračního cyklu dochází k porovnání nejmenší a největší chyby, pokud jsou přibližně stejné, je algoritmus ukončen, protože se populace dostala do minima a nemá smysl dál pokračovat v prohlédávání. Na konci metody *TrainNeuralTree* jsou pracovnímu stromu *workTree* nastaveny parametry metodou *SetParameters* na hodnoty pole *globalBestPosition*, což je nejlepší nalezená pozice (nejlepší parametry neuronového stromu). Dále je jedinci *individual* z parametru metody nastaveno nové DNA zavoláním metody *GetDna* na pracovní strom *workTree* a chyba na hodnotu proměnné *globalBestError*, která představuje chybu nejlepší částice. Poté je ještě přidán čas do vlastnosti *LearningTime* statické třídy *Info* pro získání celkové doby učení algoritmu FNT, doba je získána objektem *Stopwatch*.

Třída *MpiDifferentialEvolution* implementuje paralelní verzi algoritmu popsaný v kapitole 2.4.4. Tato třída obsahuje veřejné vlastnosti pro nastavení algoritmu:

- **PopulationSize** - udává počet jedinců v populaci, tento počet musí být násobkem spuštěných procesů, aby byl algoritmus efektivní. Datový typ je vlastnosti je *Integer*.

- **Iterations** - tato vlastnost udává počet generací algoritmu Diferenciální Evoluce. Datový typ je *Integer*.
- **RangeMin, RangeMax** - tyto vlastnosti udávají minimální a maximální hodnotu parametrů jedince, omezují hodnoty prvků polí, které představují paramery jedinců, omezují tedy prohledávaný prostor. Jejich datový typ je *Double*.
- **CrossoverRate** - tato vlastnost udává, jak moc má docházet ke křížení. Datový typ je *Double*.
- **MutationRate** - vyšší hodnoty této vlastnosti zajišťují rozdíly v populaci jedinců. datový typ je *Double*.
- **VariableMutation** - tato vlastnost je datového typu *Boolean* a udává, zda má být použita proměnná mutace.
- **MutationMin, MutationMax** - tyto vlastnosti slouží k ohraničení hodnot, kterých může nabývat mutace, pokud je hodnota vlastnosti *VariableMutation* nastavena na hodnotu *true*. Datové typy těchto vlastností jsou *Double*.

Třída implementuje metodu *TrainNeuralTree* z rozhraní *ITeacher*, jako u třídy *MpiParticleSwarm*, která má dva vstupní parametry, jeden je typu *Individual* a druhý typu *Intracommunicator*, ten je použit k MPI komunikaci. Z objektu jedince předaným parametrem se nechá vytvořit pracovní neuronový strom *workTree* typu *NeuralTree* pomocí statické metody *CreateTree* třídy *NeuralTree*. Následně je zavolána metoda *InitializeComputeNodes*, která pošle číslo operace s hodnotou 2 metodou *Broadcast* a poté pošle zkrácenou verzi struktury DNA z pracovního stromu metodou *GetShortDna*. Provádí tedy to samé jako stejnojmenná metoda ve třídě *MpiParticleSwarm*, slouží tedy k inicializaci struktury neuronového stromu na ostatních procesech, které budou dále očekávat přijetí parametrů pro tuto strukturu. Po volání metody *InitializeComputeNodes* následuje volání metody *Initialize*, které se předá parametrem pole typu *double* představující parametry jedince a to jedince pracovního stromu *workTree* typu *NeuralTree*. Aby bylo z tohoto pracovního stromu možné získat parametry, je na něj zavolána metoda *GetParameters*. Metoda *Initialize* poté vytvoří dvourozměrné pole *treeVectors* a *trialVectors* s prvním rozměrem o velikosti populace jedinců uloženém v parametru *populationSeize* a druhý rozměr o velikosti kolik má pracovní strom parametrů, hodnota se dá získat z velikosti pole *initialParameters* předaném parametrem vlastností *Length*. Poté je zavolána metoda *Evaluate*, tu popíši později protože funguje obdobně jako stejnojmenná metoda v třídě *MpiParticleSwarm*, ale tato přebírá parametrem populaci jedinců a vypočtené chyby vrací jako pole *double* hodnot, je to z důvodu, že se v algoritmu diferenciální evoluce při inicializaci ohodnocují náhodní jedinci a poté při průběhu algoritmu se ohodnocují zkušební jedinci. Chyby jsou uloženy do proměnné *treeErrors*.

Následuje hlavní cyklus diferenciální evoluce, který má maximálně tolik generací, kolik je specifikováno proměnnou *maxGenerations* nastavitelnou vlastností *MaxGenerations*. V tomto cyklu je na začátku zkontrolováno zda je nastaven parametr proměnné mutace

variableMutation na hodnotu *true*. Pokud je podmínka splněna, je zvolena mutace v parametru *mutationRate* náhodně v rozmezí *mutationMin* a *mutationMax* opět nastavitelnými obdobnými vlastnostmi. Poté je zjištěn index, na kterém se nachází nejnižší chyba, který je uložen do proměnné *bestIndex*. Dále je zahájen cyklus procházející každého jedince populace, který tvoří prvního rodiče. Uvnitř cyklu je zkontrolováno zda jedinec není na stejném indexu jako je nejlepší jedinec s nejnižší chybou *bestIndex*. Pokud je to jiný jedinec, tak jsou vybráni tři rodiče. Do dvourozměrného pole *parents* s prvním rozměrem o velikosti 3. Jako první rodič v tomto poli je vybrán jedinec s nejmenší chybou, který je na indexu *bestIndex* a tento index je uložen do pole *indexes* pro kontrolu, zda rodiče mají různé indexy. Následuje cyklus *while* ve kterém se *Rank* výběrem vybírají rodiče a kontrolují se zda rodič s vybraným indexem se nenachází již v poli *indexes* nebo není index shodný s aktuálně procházeným jedincem. Po vybrání tří rodičů je vytvořen šumový vektor, pole *double* hodnot o velikosti *dimension*, podle rovnice 17. Po vytvoření šumového vektoru je potřeba zkontrolovat, zda hodnoty šumového jedince se nachází v přípustných mezích. To provádí metoda *CheckBoundary*, která prochází pole předaném v parametru a kontroluje, zda jsou hodnoty v mezích ohraničeném vlastnostmi *RangeMin* a *RangeMax*. Pokud překročí hodnota pole tuto mez, je hodnota na daném indexu zvolena náhodně v těchto mezích. Ke generování hodnoty v mezích je použita statická metoda *GetRandomNumber* třídy *RandomGenerator*. Po upravení parametrů šumového jedince, je do pole *firstParent* zkopírován aktuálně procházený jedinec. Následně je tvořen zkušební jedinec do matice *trialVectors* první rozměr má index stejný jako index jedince v proměnné *firstParent*. Parametry zkušebního vektoru jsou ale voleny tak, že se vygeneruje pro každý rozměr dimenze náhodné číslo v intervalu $\langle 0, 1)$. Pukud je toto číslo menší jak konstanta křížení uložená v proměnné *crossoverRate*, tak je zvolena pro tuto dimenzi hodnota z šumového vektoru, pokud je náhodné číslo vetší jak proměnná *crossoverRate*, tak je zvolena hodnota z prvního rodiče *firstParent*. Po ukončení cyklu procházející populaci, který takto vytvořil novou populaci zkušebních jedincu *trialVectors*, dochází k jejich ohodnocení metodou *Evaluate* a chyby které jsou vráceny, jsou uloženy do pole *trialErrors*. Poté následuje cyklus který porovnává chybu zkušebních jedinců v poli *trialErrors* s odpovídajícími chybami jedinců předešlé generace v poli *treeErrors*. Pokud je chyba zkušebního jedince na daném indexu nižší, než je chyba na stejném indexu jedince předešlé generace, tak je na tento jedinec z předešlé generace nahrazen zkušebním jedincem a i chyba zkušebního jedince nahrazuje původní chybu. Na konci cyklu generace je podmínka, která ověřuje, zda absolutní hodnota rozdílu nejmenší a největší chyby je menší než velmi malé číslo $10^{-}14$, pokud je tato podmínka splněna, je cyklus tvorby generací ukončen příkazem *break* z důvodu, že populace jedinců zkonvergovala k minimu. Po ukončení cyklu tvorby generací, je na konci metody *TrainNeuralTree* nalezen index nejlepšího jedince nalezením nejnižší chyby z pole *treeErrors*. Následně se pracovnímu stromu *workTree* nastaví parametry nejlepšího jedince metodou *SetParameters* a předáním ji tohoto jedince formou pole z matice *treeVectors*. Nakonec je ještě aktualizován jedinec *individual* z parametru metody *TrainNeuralTree* tak, že jeho vlastnost *Dna* je nastavena na DNA pracovního stromu získaného přes metodu *GetDna*. Chyba skrz vlastnost *Error* je jedinci *individual* nastavena na hodnotu z pole *treeErrors* na pozici nejlepšího jedince. Tímto je diferenciální evoluce

ukončena a pokud došlo k nalezení lepších parametrů neuronového stromu předaného formou jedince parametrem metody *TrainNeuralTree*, tak je tento jedinec aktualizován, aby obsahoval nové parametry.

Tyto paralelní verze algoritmů spolupracují se třídou *ComputeNode*, která slouží k výpočtu chyby neuronových stromů na Slave procesech. Komunikace se Slave procesem probíhá v metodách *InitializeComputeNodes* a *Evaluate* algoritmů pro optimalizaci parametrů. Metoda *InitializeComputeNodes* pošle metodou *Broadcast* objektu *IntraCommunicator* číslo typu *Integer*, které znamená operaci, která se má provést na Slave procesech. Operací je několik typů:

- **0**- Pokud je hodnota 0, bude Slave proces očekávat přijetí jedinců z algoritmu GP
- **1**- Pro tuto hodnotu bude Slave proces očekávat přijetí parametrů jedinců z algoritmu optimalizace parametrů (PSO, DE)
- **2**- Pro tuto hodnotu bude Slave proces očekávat přijetí struktury DNA, bez zakódovaných parametrů
- **jiné**- Pro jinou hodnotu bude Slave proces ukončen

Pošle se tedy číslo s hodnotou 2. Tato operace očekává přijetí struktury neuronového stromu. Tento způsob jsem zvolil, aby se struktura nemusela posílat každou iteraci algoritmu učení. Metodou *Broadcast* je poslána struktura formou DNA řetězce typu *String* na ostatní procesy. Kratší verze DNA se dá získat metodou *GetShortDna* třídy *NeuralTree*.

Metoda *Evaluate* potom slouží k posílání parametrů jedinců u DE nebo pozic částic u PSO na všechny procesy, kde z nich vytvořeny neuronové stromy a následně ohodnoceny. Z počátku je potřeba rozdělit populaci jedinců na počet dílů odpovídajících počtu procesů. Je vytvořena trojrozměrná matice, kde první rozměr znamená pro který proces jsou data určena, druhý rozměr je jedinec o kterého se jedná a třetí rozměr jsou parametry jedince. Poté se v metodě *Evaluate* posílá metodou *Broadcast Integer* číslo s hodnotou 1, pro oznámení Slave procesu, že budou posílány parametry učícího algoritmu. Poté je posláno číslo opět metodou *Broadcast*, které znamená index výstupního pole trénovací množiny získaného z vlastnosti *OutputIndex* statické třídy *MpiTask*. Následně je zavolána metoda *Scatter*, kterou se posílá třírozměrné matice parametrů. Tato metoda vrací dvourozměrné pole i na tomto *Master* procesu, což je část populace k ohodnocení. Proto je následně vypočtena chyba tak, že se každý řádek této matice nastaví jako parametry neuronového stromu metodou *SetParameters* a zavolána na tento strom metoda *CalculateError*. Vypočtené chyby se ukládají do pole. Metodou *Gather* objektu *Intracommunicator* jsou následně tyto chyby poslány a zároveň přijaty všechny chyby ze všech procesů.

4.4.5 Třídy *ComputeNode*, *HeadNode* a *Individual*

Třída *Individual* slouží při posílání zpráv, její instance jsou serializované objekty, které obsahují *string* vlastnost *Dna* a chybu neuronové sítě *Error* typu *double*. Třída musí být se-

rializovaná z důvodu použití MPI.NET knihovny, serializované objekty je možné použít u metod pro posílání zpráv mezi procesy. Objekty této třídy používají algoritmy optimalizace struktury a parametrů. Třída *Fnt* o které bude zmíněno dále vytvoří instanci třídy *HeadNode*, pokud běží proces na Master uzlu, nebo vytvoří instanci třídy *ComputeNode*, která běží na Slave uzlích. Objekt třídy *ComputeNode* slouží k přijímání populace neuronových stromů a jejich ohodnocení tím, že pro ně vypočítají chybu na přednastavenou trénovací množinu *MpiTask* statické třídy. Tato třída bere v úvahu, kdo žádá o ohodnocení neuronových stromů. Pokud se jedná o algoritmus optimalizace striktury, tak je zbytečné posílat celé jedince formou objektů třídy *Individual*, protože se učení provádí nad stejným neuronovým stromem, jen se mění parametry sítě. Stačí proto poslat před začátkem učícího algoritmu zkrácený DNA řetězec typu *String*, který neobsahuje parametry ale jen strukturu stromu. U optimalizace struktury je potřeba přijmout celé jedince, proto je použit objekt *Individual*.

V metodě *Run* třídy *ComputeNode* je nekonečný cyklus, ve kterém je nejprve přijmuto metodou *Broadcast* objektu *Intracommunicator* číslo typu *Integer*, pokud je jeho hodnota 0, dojde k optimalizaci struktury, pokud je toto číslo 1, dojde k optimalizaci parametrů, 2 slouží k přijetí zkrácené verze DNA typu *String* před zahájením optimalizace parametrů a to metodou *Broadcast*. Pokud jiné zavolá se příkaz *break* a ukončí cyklus, to vede k ukončení procesu na Slave uzlech. Kód pro optimalizaci struktury a parametrů funguje podobně. Nejprve je metodou *Broadcast* přijato číslo udávající aktuálně zpracovávaný výstup trénovací množiny formou indexu. Poté je metodou *Scatter* přijata populace jedinců (objekty typu *Individual* nebo u optimalizace parametrů pole typu *Double*). Poté dojde k výpočtu chyby neuronových sítí a následně metodou *Gather* jsou výsledky poslány zpět na Master proces.

Třída *HeadNode* implementuje metodu *Run*. Ta v cyklu pro každý výstup trénovací množiny hledá neuronový strom ve vnitřním cyklu. Tento vnitřní cyklus má tolik iterací, kolik je definováno ve třídě *MpiTask* počet epoch FNT algoritmu. Uvnitř tohoto cyklu probíhá hledání vhodné struktury třídou *MpiGeneticProgramming* metodou *RunEvolution* a učení třídou implementující rozhraní *ITeacher* metodou *TrainNeuralTree*. Pokud je v dané epoše nalezeno řešení s menší chybou než je definované ve třídě *MpiTask*, tak je hledání přerušeno a řešení přidáno do listu, který obsahuje neuronový strom pro každý zpracovávaný výstup.

4.4.6 Třídy *DnaHelper*, *DnaFactory*, *MpiGeneticProgramming* a rozhraní *IStructure*

Rozhraní *IStructure* definuje jak má vypadat třída, která bude sloužit k nalezení vhodné struktury neuronového stromu. Je několik algoritmů k nalezení vhodné struktury neuronového stromu jako jsou PIPE, GP a další. Proto rozhraní definuje metodu *FindStructure*, která bude sloužit jako hlavní metoda k nalezení vhodné struktury. Tato metoda nevrací žádnou hodnotu a ani nemá žádné vstupní parametry. Rozhraní obsahuje dále tři vlastnosti. První důležitá vlastnost se nazývá *Communicator* typu *Intracommunicator*, která umožňuje pouze nastavit objekt tohoto typu, aby mohla třída implementující toto roz-

hraní využívat MPI komunikaci s ostatními procesy. Další vlastnost se nazává *BestTree*, která slouží k vrácení nejlepšího neuronového stromu skrz objekt typu *Individual*, který byl algoritmem nalezen. A poslední vlastností je vlastnost *TreeToTrain*, která vrací opět neuronový strom skrz objekty typu *Individual*, ale nemusí to být nejlepší jedinec, je to z důvodu, aby tu byla možnost vyzkoušet jak bude algoritmus fungovat, když se bude učit nejlepší jedinec, jedinec náhodný a nebo bude použita jiná forma strategie.

Třída *DnaHelper* obsahuje několik důležitých metod pro práci s DNA řetězcem, které je specificky zakódováno viz rovnice 11 nebo zkráceně ve formě viz rovnice 12. Tyto metody používa především třída *MpiGeneticProgramming*, která slouží k optimalizaci struktury neuronového stromu.

Důležitou pomocnou metodou této třídy je metoda *GetNumberLength*, která vrací počet znaků čísla zakódovaného v DNA od určité pozice. Vrací tedy *Integer* číslo, parametry metody jsou DNA typu *String* a index ukazující na znaky $+, x, w, a, b$, po kterých následuje číslo. Na začátku metody dochází ke kontrole, zda index předaný parametrem ukazuje na pozici v DNA na nějaký z předešlých znaků. Poté je inicializován parametr *length* typu *Integer* na hodnotu 1. Následuje cyklus ve kterém se inkrementuje hodnota *length* a který prochází maximálně do délky DNA řetězce, uvnitř cyklu dochází ke kontrole zda se na pozici *startIndex* + *length* nachází nějaký z výše popsaných znaků, pokud ano je cyklus ukončen a metoda vrátí hodnotu proměnné *length* sníženou o 1, zkrácenou verzi metody bez počáteční kontroly je možné vidět ve výpisu 13.

```
public int GetNumberLength(string dna, int startIndex)
{
    char[] characters = new char[] { '+', 'x', 'w', 'a', 'b' };
    int length = 1;
    for (; startIndex + length < dna.Length; length++)
    {
        if (characters.Contains(dna[startIndex + length]))
            break;
    }
    return length - 1;
}
```

Výpis 13: Metoda GetNumberLength třídy DnaHelper

Další často používanou metodou je metoda *GetNumber*, ta využívá výše popsanou metodu *GetNumberLength*. Tato metoda je generická, slouží k získání čísel různých datopvých typů, především však typů *double* a *Integer*. Vstupní parametry metody jsou *dna* typu *String* a *startIndex* typu *Integer*. Tato metoda tedy slouží k získání hodnoty čísla, které následuje za jedním ze znaků $+, x, w, a, b$. Využívá k tomu třídu *Convert* a její metodu *ChangeType* a to z důvodu, aby bylo možné použít při parsování čísla nezávislou kulturu, metoda je ve výpisu 14.

```
public T GetNumber<T>(string dna, int startIndex)
{
    CultureInfo provider = CultureInfo.InvariantCulture;
    int numberLength = GetNumberLength(dna, startIndex);
    T number = (T)Convert.ChangeType(dna.Substring(startIndex + 1, numberLength), typeof(T),
        provider);

    return number;
}
```

Výpis 14: Metoda GetNumber třídy DnaHelper

Metoda *GetParameterIndexes* slouží k vracení pozic znaků definovaného parametrem metody z DNA řetězce. Může být použita k vrácení indexů například + znaků, tedy začátků neuronů v neuronovém stromě. Tato metoda prochází DNA řetězec typu *String* a každý výskyt znaku typu *char* zaznamenává do generické kolekce typu *List Integer* hodnot, tuto kolekci následně metoda vrací.

Velice používanou metodou je metoda *GetSubDnaLength*. Ta slouží k zjištění délky řetězce podstromu v DNA. Je pak možné tento podstom nahradit jiným nebo jej odstranit z DNA. Tato metoda má marametry *dna* typu *String* a *startIndex* typu *Integer*. Metoda vrací délku ve formě čísla typu *Integer*. Metoda očekává přijetí DNA řetězce a indexu odkazující na znak +, kterým začíná definice neuronu jako kořenu stromu se všemi připojenými neurony a jejich parametry, nebo znak *x*, který odkazuje na definici vstupního neuronu. Metoda funguje rekurzivně, zpočátku je ověřeno, zda parametr *startIndex* odkazuje odkayuje v DNA na znak *x* nebo +. Pokud je tato podmínka splněna, tak dochází k zjištění na jaký znak odkazuje, pokud totiž odkazuje na *x*, což znamená, že následuje vstupní neuron, tak metoda vrátí délku čísla za tímto znakem zvětšenou o 1 za délku znaku. Jedná se o ukončovací podmínku v rekurzi. Pokud *startIndex* odkazuje na znak +, tak jsou očekávány parametry neuronu. Je tedy inicializována proměnná *length* na hodnotu délky čísla za tímto znakem s přičtenou jedničkou za + znak. K délce čísla je použita metoda popsaná výše. Je i zjištěna hodnota čísla za *x* znakem, aby bylo možné určit kolik má uzel potomků. Dále je přičtena k proměnné *length* délka čísla parametru *a* a *b* se znaky. Poté následuje cyklus procházející potomky. V tomto cyklu jsou je přidán do proměnné *length* číslo jedna jako znak váhy a délka čísla váhy. Poté je přičtena k této proměnné hodnota, co vrátí rekurzivní volání metody *GetSubDnaLength* s parametrem DNA a indexem takovým, že k proměnné *startIndex* je přičtena dosavadní délka v proměnné *length*. Kód zkrácené verze metody bez kontroly vstupu je možné vidět ve výpisu 15. Na tuto metodu navazuje metoda *GetSubDna*, která má stejné parametry, ale vrací *String* řetězec, funguje tak, že zjistí délku *length* podřetězce DNA začínajícím znaky + nebo *x*. Následně vrátí podřetězec DNA metodou *Substring* s prvním parametrem DNA ze vstupu metody a druhým parametrem *length*.

```
public int GetSubDnaLength(string dna, int startIndex){
    if (dna[startIndex] == '+') {
        int length = 1 + GetNumberLength(dna, startIndex);
        length += 1 + GetNumberLength(dna, startIndex + length);
        length += 1 + GetNumberLength(dna, startIndex + length);
        int children = GetNumber<int>(dna, startIndex);
        for (int i = 0; i < children; i++){
            length += 1 + GetNumberLength(dna, startIndex + length);
            length += GetSubDnaLength(dna, startIndex + length);
        }
        return length;
    }
    if (dna[startIndex] == 'x')
        return 1 + GetNumberLength(dna, startIndex);
    return 0;
}
```

Výpis 15: Metoda GetSubDnaLength třídy DnaHelper

Velice důležitou metodou pro genetické programování je metoda *GetChildrenNonLeafNodesIndexes*. Tato metoda má jako vstupní parametry *dna* typu *String* a *startIndex* typu *Integer*. Metoda vrací kolekci *List* indexů ukazujících na pozice neuronů, kteří jsou potomky neuronu definovaného indexem *startIndex* a nejsou vstupními neurony. V algotimu genetického programování je možné tuto metodu využít při mutaci. Ukázka kódu této metody je ve výpisu 16. V této ukázce chybí ale počáteční kontrola vzda index ukazuje na pozici v *dna* parametru a zda ukazuje na znak +, definující kořenový neuron.

```
public List<int> GetChildrenNonLeafNodesIndexes(string dna, int startIndex){
    List<int> indexes = new List<int>();
    if (dna[startIndex] == '+') {
        int length = 1 + GetNumberLength(dna, startIndex);
        length += 1 + GetNumberLength(dna, startIndex + length); //for 'a' parameter
        length += 1 + GetNumberLength(dna, startIndex + length); //for 'b' parameter
        int children = GetNumber<int>(dna, startIndex);
        for (int i = 0; i < children; i++){
            length += 1 + GetNumberLength(dna, startIndex + length);
            if (dna[length] == '+') {
                indexes.Add(length);
            }
            length += GetSubDnaLength(dna, startIndex + length);
        }
    }
    return indexes;
}
```

Výpis 16: Metoda GetChildrenNonLeafNodesIndexes třídy DnaHelper

Další podobně fungující metodou, jako je metoda popsaná výše, je metoda *GetChildrenLeafNodesIndexes*. Tato metoda má stejné parametry a také vrací kolekci ukazující na pozice potomků. Tentokrát ale metoda vrací pozice pouze vstupních neuronů. Opět je to důležitá metoda v genetickém programování při mutaci DNA řetězce. Tato metoda

se liší od výše popsané metody *GetChildrenNonLeafNodesIndexes* pouze ve vnitřní podmínce, kdy pro přidání indexu do kolekce se kontroluje na znak *x*. Stejný princip používá i metoda *GetChildrenNodesIndexes* Ta už nekontroluje vnitřním cyklem co přidat, ale vrací všechny potomky spojené s kořenovým neuronem.

V genetickém programování jsou potřebné i metody k tvorbě neuronových stromů, ať už při mutaci nebo při tvorbě náhodné populace. K tomuto slouží statická třída *DnaFactory*, která vytváří tyto stromy. Spolupracuje se statickou třídou *MpiTask*, z důvodu inicializace parametrů těchto nových stromů a určení jak široké stromy mají být. To je zajištěno náhodně v zadaných mezích. Meze je možné ovlivnit vlastnostmi statické třídy *MpiTask*. Vlastnosti *WeightMax* a *WeightMin* slouží při generování hodnoty vah v tomto definovaném rozmezí. Vlastnosti *Amax* a *Amin* slouží stejně ale pro parametr *a* aktivační funkce a vlastnosti *Bmax* a *Bmin* slouží obdobně pro parametr *b*. Třída *DnaFactory* má dvě metody ke generování náhodných stromů. První se jmenuje *CreateRandomInitialTree*, vrací řetězec DNA typu *String* a nemá žádné vstupní parametry.

Další metodou třídy *DnaFactory* je *CreateOneNeuronTree*, která vrací řetězec DNA typu *String* a to neuronového stromu s jedním neuronem a náhodně zvoleným počtem vstupních neuronů, jedná se tedy o strom bez neuronů ve skryté vrstvě. První jsou zvoleny náhodně hodnoty parametrů aktivační funkce kořenového neuronu v rozmezí zmíněném výše. Tento neuron musí mít minimálně jeden vstupní neuron, proto je náhodně zvoleno číslo odpovídající indexu vstupu v trénovací množině, interval jde od nuly po počet vstupů trénovací množiny. Následně je vytvořeno pole *inputs* typu *Boolean* o velikosti počtu vstupů a na tento náhodně vybraný index je uložena hodnota *true* zbytek jsou hodnoty *false*. Poté je vygenerováno číslo v intervalu $\langle 0, 1)$ a uloženo do proměnné *selectRate* typu *Double*. Dále metoda pokračuje inicializací proměnné *rootInputs* typu *Integer*, která počítá počet vstupních neuronů a cyklem, který prochází pole *inputs*. Uvnitř cyklu dochází k ověření podmínou, zda hodnota pole na procházeném indexu je *false*. Pokud je tato podmínka splněna je ověřeno další podmínkou, zda náhodně vygenerované číslo v intervalu $\langle 0, 1)$ je menší než hodnota proměnné *selectRate*. Pokud je i tato podmínka splněna, dochází k inkrementování proměnné *rootInputs* a na procházený index v poli *inputs* je nastavena hodnota *true*. Tímto je zjištěno, jaké vstupní neurony budou spojeny s kořenovým neuronem. Následuje tedy tvorba řetězce DNA. Je vytvořen objekt *build* třídy *StringBuilder*. Tomuto objektu je přidán řetězec začínající znakem "+", za tímto znakem následuje hodnota proměnné *rootInputs*, poté následují znaky *a* s vygenerovanou hodnotou a *b* taky s vygenerovanou hodnotou typu *Double*. Tímto je vytvořen začátek DNA řetězce, poté je potřeba ještě přidat vstupní neurony s odpovídajícími váhami. V cyklu se prochází pole *inputs* a pokud je hodnota tohoto pole na procházeném indexu *true*, tak je na objekt *build* zavolána metoda *Append* a jako parametr je této metodě předan řetězec např. ve formě *w0.2x1*. Hodnota váhy je ale zvolena náhodně v inicializačním rozmezí *wMin* a *wMax*. Číslo vstupu je procházený index *inputs* pole. Metoda nakonec vrací vytvořený řetězec zavoláním metody *ToString* na objekt *build*.

Pro tvorbu počáteční populace neuronových stromů v Genetickém Programování slouží metoda *CreateRandomInitialTree* třídy *DnaFactory*. Tato metoda nechá vytvořit kořenový neuron s náhodným počtem vstupů, jako metoda *CreateOneNeuronTree*, dále se liší ale v tom že je vygenerováno náhodné číslo od jedné do hodnoty vlastnosti *maxInitialChildrenNeurons* statické třídy *MpiTask*, toto číslo je uloženo do proměnné *nonTerminalsCount* a je přičteno k proměnné *rootInputs*, výsledná hodnota znamená počet vstupů do kořenového neuronu. Poté je vytvořen začátek DNA ve formě např. +8a0.11b0.6, následuje přidávání vstupních neuronů podle pole *inputs* stejně jako v předešlé metodě. Nakonec ale dochází k přidávání vnitřních neuronů ke kořenovému a to tak, že v cyklu se přidávají ke stávající DNA náhodně vygenrované váhy v rozmezí *wMin* a *wMax* a k těmto vahám se nechá vytvořit metodou *CreateOneNeuronTree* vnitřní neuron. Cyklus probíhá tolikrát, kolik je hodnota proměnné *nenTerminalCount*. Výsledný DNA řetězec je vrácen zavoláním metody *ToString* na objekt *StringBuilder*.

Poslední a nejdůležitější třídou v tomto jmenném prostoru je třída *MpiGeneticProgramming* implementující paralelní verzí algoritmu Genetického Programování popsaný v sekci 2.3.1. Tato třída obsahuje několik vlastností pro nastavení algoritmu:

- **MaxGnerations** - udává kolik generací se má provést, volba této hodnoty záleží na řešeném problému, datový typ této vlastnosti *Integer*.
- **RequiredError** - udává přípustnou chybu neuronového stromu, pokud nejlepší neuronový strom má menší hodnotu chyby, je algoritmus ukončen, datový typ této vlastnosti je *Double*.
- **PenaltyRate** - tato vlasnost slouží k přepočtu chyby neuronového stromu na zdatnsot neuronového stromu. Je typu *Double* a zhoršuje zdatnost oproti chybě v závislosti na velikosti neuronového stromu. Čím větší je toto číslo, tím větší je rozdíl ve zdatnostech neuronových stromů se stejnou chybou ale různou velikostí.
- **EliteSize** - vlasnost udává velikost elitní populace, musí být podmnožinou velikosti populace. Tento počet nejlepších jedinců má zaručen, že nebude ztracen a bude i v další generaci. Datový typ je *Integer*.
- **PopulationSize** - udává kolik jedinců se nachází v populaci. Větší počet umožňuje lépe prohledat prostor řešení. Datový typ je *Integer*.
- **CrossoverRate** - hodnota této vlastnosti určuje, jak často má být provedeno křížení při vytváření nové populace. Pokud je hodnota této vlastnosti jedna, tak dochází pouze ke křížení a ne k reprodukci, datový typ této vlastnosti je *Double*.
- **MutationChance** - tato vlastnost určuje šanci na mutaci jedince, který byl vytvořen při křížení. Pokud je hodnota jedna, tak každý vytvořený jedinec je zmutován, datový typ je *Double*.

V konstruktoru této třídy jsou nastaveny různé vlastnosti na výchozí hodnoty a je zavolána metoda *InitializeNewPopulation*. Tato metoda slouží k vytvoření počáteční populace jedinců a to tak, že je vytvořena kolekce *population* generického typu *List* jedinců *Individual*. Tato kolekce je naplněna cyklem jedinci. Jedinci jsou vytvořeni jako objekty třídy *Individual* a těmto objektům při vytváření je konstruktorem předáno DNA. DNA je vytvořeno metodou *CreateRandomInitialTree* třídy *DnaFactory*. Při tvorbě nové populace je i inkrementována hodnota vlastnosti *StructureTreesCreated* statické třídy *Info*. Na konci metody je zavolána metoda *Sort* pro setřízení populace podle jejich zdatnosti.

Metoda *Sort* setřizuje populaci jedinců *population*, která je globálně deklarována ve třídě. Na kolekci jedinců je zavolána "Extension"metoda *OrderBy*. K určení jak je jedinec vhodný, je použita metoda *Fitness*, ta má vstupní parametr typu *Individual* a vrací datový typ *Double*. Tato metoda provádí přepočet chyby neuronového stromu na zdatnost podle počtu neuronů neuronového stromu, počet neuronů jeupraven pomocí proměnné *penaltyRate*. Obě metody jsou privátní. Metody je možné vidět ve výpisu 17.

```
private void Sort()
{
    population = population.OrderBy(x => Fitness(x)).ToList() ;
}
private double Fitness(Individual individual)
{
    double error = individual.Error;
    int neurons = dnaHelper.GetParameterIndexes(individual.Dna, '+').Count;
    neurons += dnaHelper.GetParameterIndexes(individual.Dna, 'x').Count;
    double fitness = error;

    fitness *= 1 + neurons * penaltyRate;
    return fitness;
}
```

Výpis 17: Metody Sort a Fitness třídy MpiGeneticProgramming

Metodou která odstartuje hledání řešení se nazývá *RunEvolution*, ta provádí vykonávání hlavního algoritmu Genetického Programování. Metoda vrací referenci na populaci jedinců ve formě generické kolekce typu *List* složené z objektů *Individual*. V této metodě je spuštěn základní cyklus procházející jednotlivé generace. Uvnitř cyklu je zavolána metoda *CreateNewGeneration*, která vytváří ze stávající populace populaci do další generace. Poté je populace ohodnocena zavoláním metody *Evaluate*, ta paralelně spočte chyby jednotlivých neuronových stromů. Následuje volání metody *Sort*, která setřídí populaci podle zdatnosti jedinců. Takto probíhá cyklus až do generace specifikované v proměnné *maxGeneration* nebo dokud není nalezen neuronový strom s lepší než požadovanou chybou. Doba běhu tohoto cyklu je zaznamenávána pomocí objektu *Stopwatch* a je potom přičtena do vlastnosti *StructureTime* statické třídy *Info* v sekundách. Na konci je vrácena reference na populaci jedinců pro možnost učení.

Metoda *CreateNewGeneration*, která byla zmíněna v předešlém odstavci, tedy generuje novou generaci jedinců. První je inicializonána nová populace jako kolekce typu *List* do proměnné *newGeneration*. Do této nové generace jsou první přidáni jedinci ze stávající generace cyklem. Jejich počet je nastaven v proměnné *eliteSize*. Poté následuje další cyklus dokud nová generace nemá velikost, jaká je specifikována v proměnné *populationSize*. V tomto cyklu je vygenerováno náhodné číslo v intervalu $\langle 0, 1)$ a uloženo do proměnné *selection* typu *Double*. Následuje podmínka která vyhodnocuje, zda dojde ke křížení nebo k reprodukci. Pokud je náhodně vygenerované číslo v proměnné *selection* menší jak hodnota v proměnné *crossoverRate*, dojde ke křížení, jinak dojde k reprodukci. Při křížení dochází k pořadovému výběru (rank selection) rodičů, tento výběr je simulován rovnicí autora D. Whitley [15] implementované ve výpisu 18, tato rovnice vrací index s hodnotou v intervalu $\langle 0, population.Count)$ s tím, že upředňostňuje počáteční indexy a je ovlinvněna parametrem *c*, ten je možné nastavit v intervalu $(0, 2\rangle$, se zvyšující se hodnotou parametru *c* se zvyšuje pravděpodobnost výskytů prvního indexu oproti poslednímu indexu. Takto jsou vybráni oba rodiče a následně je provedena metoda *Crossover*, která provádí křížení těchto dvou rodičů a vrací dva DNA řetězce potomků. Poté je pro každého potomka zjištěno, zda dojde k jeho mutaci. Je vygenerováno číslo opět v intervalu $\langle 0, 1)$ a pokud je menší jak hodnota proměnné *mutationRate*, tak dojde k mutaci jedince. Mutace je provedena zavoláním metody *Mutation* a předáním DNA jedince. Po možné mutaci jsou vytvořeny objekty *Individual* z pomocí DNA řetězce a tyto objekty jsou přidány do kolekce nové generace *newGeneration*. Po přidání je i inkrementována hodnota vlastnosti *StructureTreesCreated* statické třídy *Info*. U přidání druhého potomka dochází ke kontrole, zda počet jedinců v nové generaci nepřesáhl velikost populace. Pokud generace přesáhla velikost populace, tak druhý potomek není přidán do nové generace. Pokud dojde k reprodukci, tak je pořadovým výběrem vybrán jedinec, který může být dále zmutován jako při křížení a poté je přidán do nové generace. Na konci metody je aktuální generace nahrazena novou generací jedinců.

```
Individual father = null;
double c = rankSelectionBias;
double random;
int index;
random = RandomGenerator.GetRandomNumber(0, 1);
index = (int)Math.Floor(population.Count / (2.0 * (c - 1)) * (c - Math.Sqrt(c * c - 4 * (c - 1)
    * random)));
father = population[index];
```

Výpis 18: Pořadový výběr (Rank Selection)

Metoda která provádí křížení dvou jedinců se nazývá *Crossover*. Tato metoda má čtyří parametry typu *String*. Dva parametry slouží k předání rodičovských DNA řetězců. Další dva parametry jsou označeny klíčovým slovem *out* a představují DNA řetězce potomků, kteří se v této metodě vytvoří a jsou vráceni těmito parametry z metody. Metoda první získá pozice "+"znaků v DNA řetězcích rodičů pomocí metody *GetParametersIndexes* třídy *DnaHelper*. Tyto indexy jsou uloženy do kolekcí a znamenají vnitřní neurony stromu. Poté jsou vygenerována dvě náhodná čísla od 0 do počtu prvků dané kolekce. Tato čísla

znamenají indexy neuronů, které budou vybráni pro vytvoření podstromu, kde vybraný neuron představuje kořen tohoto podstromu. K získání podstromu je použita metoda *GetSubDnaLength* třídy *DnaHelper* a metoda *Substring* třídy *String*. DNA podstomu je následně odstraněno z DNA rodiče metodou *Remove* třídy *String* a na odstraněné místo je vloženo DNA podstromu druhého rodiče. Tímto je tedy provedena výměna podstromů v DNA rodičů, vzniklé řetězce jsou nastaveny výstupním parametrům této metody.

Metoda *Mutation* provádí mutaci daného jedince. Vstupním parametrem je *String* řetězec DNA nějakého jedince a metoda vrací zmutovaný řetězec. Tato metoda provádí několik typů mutace. Typ mutace je zpočátku metody vybrán náhodně. Typy mutací jsou:

- **Nahrazení vstupního neuronu jiným vstupním neuronem**
  Pokud je vybrán tento typ mutace, tak je provedeno získání pozic vstupních neuronů z DNA řetězce metodou *GetParameterIndexes* třídy *DnaHelper* a tyto pozice jsou uloženy do kolekce. Z této kolekce je vybrán náhodně jeden index a pro něj je vygenerován nový vstup. Následně jsou použity metody *Remove* a *Insert* třídy *String* pro nahrazení starého vstupu na daném indexu v DNA a jeho nahrazení novým vstupem, tedy novým vstupním neuronem.

- **Nahrazení podstromu vstupním neuronem**
  Tento typ mutace je proveden tak, že metodou *GetParameterIndexes* třídy *DnaHelper* jsou získány indexy vnitřních neuronů a tyto indexy jsou uloženy do kolekce. Pokud je v této kolekci více jak jeden prvek (první je kořenový neuron a ten není použit), tak je vybrán náhodně jeden index z této kolekce kromě prvního. Poté je na tomto indexu odstraněn celý podstrom metodou *Remove* třídy *String*, kde délka odstraněného podstromu je určena pomocí metody *GetSubDnaLength* třídy *DnaHelper*. Následně je vygenerono *Integer* číslo do velikosti proměnné *inputCounts* udávající počet vstupů trénovacího vzoru. Toto číslo se znakem *x* je pozužito jako náhrada za odstraněný podstrom a znamená vstupní neuron.

- **Nahrazení vstupního neuronu podstromem**
  Tento typ mutace se provádí tak, že jsou vloženy do kolekce indexy vstupních neuronů metodou *GetParameters* třídy *DnaHelper*. Pokud tato kolekce obsahuje alespoň jeden prvek, tak dojde k náhodnému vybrání jednoho indexu z této kolekce. Poté na daném indexu dojde k odstranění řetězce metodou *Remove* třídy *String* představující vstupní neuron a na toto místo je vložen řetězec vygenerovaného podstromu pomocí metody *CreateOneNeuronTree* třídy *DnaHelper*.

Poslední a nejdůležitější je metoda *Evaluate*, které slouží k rozeslání populace jedinců na všechny procesy, kde budou vypočteny chyby těchto jedinců, kteří jsou následně vráceni této třídě. Populace je rozdělena rovnoměrně, protože ale neuronové stromy mají jinou velikost, je možné že tato metoda nebude v synchronní verzi moc efektivní pro velice pestrou populaci jedinců, v tom případě by bylo nutné navrhnou lepší řešení, ať už asynchronní nebo rozdělit populaci tak, aby rozdíl v času mezi nejrychlejším a nejpomalejším procesem nebyl moc velký. V této metodě je vytvořena kolekce kolekcí, které obsahují

jedince pro daný proces. Poté je zavolána metoda *Barrier* třídy *Intracommunicator* a metodou *Broadcast* je poslán typ operace 0, který znamená, že se jedná o ohodnocení celého neuronového stromu. Dále je poslán metodou *Broadcast* výstupní index trénovací množiny, pro který jsou neuronové stromu tvořeny. Následuje volání metody *Scatter*, která rozešle populaci ostatním procesům, kód výpočetních procesů je ve třídě *ComputeNode*. Volání metody *Scatter* vrací část populace pro tento Master proces ve formě generické kolekce *List* jedinců *Individual*. V cyklu jsou procházeni tito jedinci a pro každého jedince je v tomto cyklu vytvořen neuronový strom metodou *CreateTree* s parametrem DNA jedince třídy *NeuralTree*. Aktuálnímu jedinci je nastavena chyba na vypočtenou chybu stromu zavoláním metody *CalculateError*. Následuje volání metody *Barrier* po tomto cyklu a následuje volání metody *Gather* pro získání zpět jedinců od všech procesů, kde v parametru jsou předáni jedinci s vypočtenou chybou daného procesu. Tato metoda vracé pole kolekcí, které je nutné spojit zpět do jedné kolekce všech jedinců. V cyklu se tedy prochází pole kolekcí a do prázdné kolekce *individuals* jsou metodou *AddRange* přidány jednotlivé části populace. Na konci metoda touto populací nahrazuje starou populaci jedinců v kolekci *population*.

4.4.7 Třídy *Connection*, *Neuron*, *InputNeuron*, *NeuralTree* , *NeuralNetwork* a rozhraní *INeuron*

Tyto třídy a rozhraní definují neuronovou síť složenou z neuronových stromů. Neuronovou síť definuje třída *NeuralNetwork*, ta obsahuje kolekci *List* neuronových stromů *NeuralTree*, tak že strom na indexu 0, odpovídá indexu 0 výstupní trénovací množiny. List neuronových stromů je předán této třídě konstruktorem. Tato třída má ještě dvě veřejuné metody. První metoda je *CalculateError*, která na danou trénovací množinu ve třídě *MpiTask* vypočte průměrnou chybu všech neuronových stromů. Typ chyby je opět definovaný ve třídě *MpiTask*, typy chyb mohou být MSE a RMSE.

Třída *NeuralTree* představuje neuronový strom. Tato třída má dvě statické metody, které umožňují vytvořit neuronový strom parametrem předaným DNA. Pro verzi DNA se zakódovanými parametry je použita metoda *CreateTree* a pro zkrácenou verzi je metoda *CreateTreeFromShortDna*, která vytvoří neuronový strom *NeuralTree* s náhodnými parametry. Tvorba stromu je implementována tak, že se vytvoří první kořenový neuron, ten se uloží do parametru *OutputNeuron* této třídy. Následně se zavolá na tento neuron odpovídající metoda *FromDna* nebo *FromShortDna*. Tyto metody dále dotváří vazby a neurony z předaného DNA formou objektu typu *StringBuilder*. Dále obsahuje třída *NeuralTree* několik veřejných metod. Metoda *CalculateError* vypočítá chybu neuronového stromu pro danou testovací množinu, vrací chybu jako číslo typu *Double*. Metody *GetDna* a *GetShortDna* vracejí DNA řetězec typu *String*. Tyto metody volají na výstupní neuron metodu *ToDna* nebo *ToShortDna*. Ty vytářejí DNA řetězec od kořene rekurzivně do hloubky. Přistupovat k výstupnímu neuronu je možné skrz vlastnost *OutputNeuron*. Další důležité metody jsou *GetParameters* a *SetParameters*. Tyto metody slouží k získání a nastavení parametrů v přesně daném pořadí. Důležitá je metoda *GetSubTree* definovaná rozhraním *INeuron*, ta totiž rekurzivně do hloubky vkládá neurony do kolekce *List*. Poté jsou nasta-

veny nebo zjištěny parametry které jsou vloženy do pole typu *Double*. Poslední veřejnou metodou je *GetResponse*, která parametrem přijme pole vstupních hodnot a zavolá metodu *GetState* výstupního neuronu, která na tento vzor vydá rekurzivně výstup stromu.

Rozhraní *INeuron* definuje několik metod, které musí imlementovat vstupní nebo vnitřní neuron. Je zde metoda *GetState*, ta vrací hodnotu na výstupu neuronu jako číslo typu *Double*. Metoda *GetSubTree* s parametrem kolekce *List* objektů *INeuron* slouží k rekurzivnímu přidávání neuronů do této kolekce. Dále jsou definovány metody *ToDna*, *ToShortDna*, *FromDna* a *FromShortDna*, všechny mají parametr typu *StringBuilder*.

Třída *Neuron* implementuje rozhraní *INeuron* a představuje vnitřní neuron neuronového stromu. V konstruktoru neuronu se nastavuje typ aktivační funkce podle statické třídy *MpiTask*, dále jsou zvoleny náhodně paramatry aktivační funkce v mezích definovaných opět třídou *MpiTask*. Metoda *GetState* prochází všechny neurony vstupující do aktualního neuronu a volá na ně metodu *GetState*, poté je provedena sumace těchto hodnot, která představuje potenciál neuronu. Nakonec je vrácen výstup neuronu tak, že je potenciál s parametry aktivační funkce přepočten daným typem aktivační funkce. Metoda *GetSubTree* funguje tak, že do kolekce *List* objektů typu *INeuron* jako parametr této metody, je přidán objekt aktuálního neuronu a následně je tato metoda zavolána na všechny neurony spojeny s tímto neuronem. Toto volání je ukončeno vstupním neuronem, který má metody implementované odlišně. Metoda *ToDna* je implementována tak, že do objektu *StringBuilder* předaný parametrem, je metodou *Append* přidáno počet vstupů do neuronu (např. *+3*), dále jsou přidány parametry aktivační funkce (např. *a0.11b0,336*), poté je přidána pro každý vstup váha (např. *w0,66*) a zavolána metoda *ToDna* vstupních neuronů. Metoda *ToShortDna* funguje stejně až na to, že nejsou do DNA řetězce přidávány parametry aktivačních funkcí a vah. Další metodou je *FromDna*, ta čte z objektu *StringBuilder* a přečtenou část odstraňuje. Přečtenou částí je počet vstupů, parametry aktivační funkce a poté v cyklu čte hodnotu váhy a volá stejnou metodu na vstupní neurony. Z přečtené části vytváří spojení a neurony. Metoda *FromShortDna* je opět velice podobné jen slouží k vytvoření neuronového stromu, s použitím DNA řetězce bez parametrů aktivačních funkcí a vah. Parametry jsou zvoleny náhodně v rozmezí definovaném třídou MpiTask.

Poslední třídou ve jmenném prostoru *Network* je *InputNeuron*. Ta implementuje rozhraní *INeuron*. Metoda *GetState* oproti stejnojmenné metodě ve třídě *Neuron* vrací vstup do neuronové sítě na indexu definovaném vlastností *Index*, vstupní pole do neuronové sítě tato metoda bere z parametru. Metoda *GetSubTree* přidává do kolekce *List* typu *INeuron* předanou parametrem aktuální objekt této třídy klíčovým slovem *this*. Metoda *ToDna* provádí přidání do objektu *StringBuilder* řetězce definující index (hodnota vlastnosti *Index* této třídy) vstupu (např. *x5*). Metoda *ToShortDna* funguje stejně. Metoda *FromDna* "odloupne"začátek řetězce pro definování objektu této třídy předaný parametrem typu *StringBuilder* a nastaví podle toho hodnotu vlastnosti *Index*. Metoda *FromShortDna* funguje až na detail v parsování stejně.

4.4.8 Třídy *MeanSquareError*, *RootMeanSquareError* a rozhraní *ICalculateError*

Rozhraní *ICalculateError* definuje, jak má vypadat třída pro výpočet chyby neuronové sítě. Toto rozhraní definuje metodu *Calculate* která má parametry typu *NeuralTree* a *TrainingSet*. V této knihovně jsou definované dva základní výpočty této chyby pomocí tříd *MeansSquareError* a *RootMeanSquareError*, které implementují toto rozhraní. V metodě *Calculate* se počítá chyba neuronové sítě tak, že se nechá vypočítat výstup pro každý vzor trénovací množiny danému neuronovému stromu a porovná se s ideální hodnoutou podle výstupu trénovací množiny, index výstupu je uložen v parametru *OutputIndex* statické třídy *MpiTask*. Porovnání je provedeno jako druhá mocnina rozdílu výstupu a ideální hodnoty. Poté je tato hodnota vydělena počtem vzorů trénovací množiny viz rovnice 9. U třídy *RootMeanSquareError* je tato hodnota ještě odmocněna viz rovnice 10. Třídy pro výpočet chyby ještě přetěžují metodu *ToString* a vracejí zde název typu výpočtu chyby např. *Mean Square*.

4.4.9 Třída *AsyncMpiGeneticProgramming*

Dodatečně byla přidána asynchronní verze GP algoritmu. Tato verze se liší metodou *Evaluate* oproti synchronní verzi reprezentované třídou *MpiGeneticProgramming*. Musela být rozšířena i třída *ComputeNode* o zpracování asynchronních zpráv. Mohla být naprogramována jiná verze třídy implementující rozhraní *IcomputeNode*, ale takto bude moct být třída *ComputeNode* použita k porovnání synchronní a asynchronní verze GP algoritmu. Asynchronní verze byla přidána z důvodu nižší efektivity synchronní verze.

Asynchronní verze funguje tak, že proces s rank hodnotou 0, provádí algoritmus GP pomocí této třídy a jak potřebuje ohodnotit chyby neuronových stromů, tak metodou *Broadcast* pošle na Slave procesy reprezentované třídou *ComputeNode* číslo *Integer* s hodnotou 4, to udává, ať Slave proces spustí blok pro asynchronní příjem neuronových stromů k ohodnocení. Populace je dále setřízena od největších stromů po nejmenší. Následuje podmínka zda počet procesů je větší jak 1, pokud ne, jsou vypočítány chyby stromů na tomto Master procesu. Pokud je více procesů spuštěných, tak je v nekonečném cyklu *while* volána metoda *ImmediateReceive*, tato metoda přijímá z jakéhokoliv zdroje hodnotu *Boolean*, která udává, zda chce Slave proces práci nebo poslat výsledek práce:

- Pokud Slave proces chce odeslat výsledek práce, je zavolána znovu metoda *ImmediateReceive*, která očekává přijetí objektu *Individual* z daného Slave procesu. Přijatý objekt *Individual* je uložen do kolekce *calculatedIndividuals*.
- Pokud Slave proces chce novou práci, je mu poslán jedinec k ohodnocení metodou *ImmediateSend*. Pokud už byli všichni jedinci ohodnoceji je mu poslán touto metodou jedinec s hodnotou DNA vlastosti *null* pro ukončení asynchronního bloku. Poté je testováno, jestli tato ukončující zpráva byla odeslána všem procesům, pokud ano, je ukončen cyklus *while* přikazem *break*. Nakonec je populace GP nastavena na tuto přijatou populaci.

4.5 Použití knihovny

Pro použití knihovny je potřeba vytvořit nový projekt typu konzolová aplikace v některém z jazyků .NET, poté je potřeba v tomto projektu přidat na tuto knihovnu referenci. K danému projektu je potřeba přidat i referenci na knihovnu MPI.NET. Ukázku použití této knihovy je možné vidět ve výpisu 19. V metodě *Main* se musí provést několik kroků ke správnému použití knihovny:

- Inicializace prostředí MPI například pomocí konstrukce *using*
- Dále je potřeba nastavit třídu *MpiTask*
    1. Nastavení vlastnosti *ErrorType* na nějaký typ výpočtu chyby implementující rozhraní *IErrorType*
    2. Nastavení vlastnosti *ActivationFunction* na nějaký typ výpočtu aktivační funkce implementující rozhraní *IActivationFunction*
    3. Nastavení vlastnosti *RequiredError* na chybu sítě, kdy se má přestat hledat lepší řešení.
    4. Nastavení vlastnosti *EpochSteps* na hodnotu maximálního počtu provedených epoch algoritmu FNT.
    5. Nastavení vlastnosti *MaxInitialChildrenNeurons*, závisí na problému, udává meximální počet neuronů spojených k neuronu ve vyšší vrstvě v nově vygenerovaných stromech.
    6. Nastavení vlastnosti *MaxAfterLearnIterations*, tato vlastnost umožní doučit nalezené řešení definovaným počtem iterací.
    7. Nastavení vlastnosti *TrainingSet*, té je nutné přidat trénovací množinu pomocí třídy *TrainingSet*.
    8. Nastavení vlastnosti *ComputeNode* na třídu implementující rozhraní *IComputeNode*, která obsahuje kód co bude provázen na Slave procesech.
    9. Nastavení vlastnosti *StructureAlgorithm* na požadovaný algoritmus optimalizující struktury, třídu implementující rozhraní *IStructure* plus jeho požadované parametry.
    10. Nastavení vlastnosti *Teacher* na požadovaný algoritmus optimalizující parametry neuronového stromu, třídu implementující rozhraní *ITeacher* plus jeho požadované parametry.
- Poté je potřeba vytvořit instanci třídy *Fnt*.
- Následovat musí podmínka, která zjišťuje *Rank* procesu pomocí komunikátoru *Fnt*. Pokud je *Rank* nenulový, je potřeba zavolat pouze metodu *FindSolutionForTask* na objekt *Fnt*. Pokud je *Rank* roven nule:

1. Je potřeba spustit vyhledávání řešení zavoláním metody *FindSolutionForTask* na instanci *Fnt*, na tomto procesu tato metoda vrací nalezené řešení jako objekt *NeuralNetwork*, se kterým je možné dále v tomto bloku kódu pracovat. Tuto metodu je možné zavolat vícekrát a pak vybrat nejlepší řešení.
2. Na konci musí být zavolání metody *SendEndMessage*, ta ukončí Slave procesy.

```
static void Main(string[] args){
    using (new MPI.Environment(ref args)){
        Intracommunicator comm = Communicator.world;
        MpiTask.ErrorType = new RootMeanSquare();
        MpiTask.ActivationFunction = new Sigmoid();
        MpiTask.RequiredError = 0.0001;
        MpiTask.MaxEpochSteps = 10;
        MpiTask.MaxInitialChildrenNeurons = 5;
        MpiTask.MaxAfterLearnIterations = 0;
        MpiTask.TrainingSet = new TrainingSet(inputsSet, outputsSet);
        MpiTask.ComputeNode = new ComputeNode();
        AsyncMpiGeneticProgramming aGp = new AsyncMpiGeneticProgramming(
            comm);
        aGp.RankSelectionBias = 2;
        aGp.MaxGeneration = 100;
        aGp.EliteSize = 0;
        aGp.UpdatePerGeneration = 100;
        aGp.PenaltyRate = 0.001;
        aGp.MutationChance = 0.2;
        aGp.CrossoverRate = 0.3;
        aGp.PopulationSize = 128;
        aGp.OnSolutionFound += testGp_OnSolutionFound;
        aGp.OnGenerationChanged += testGp_OnGenerationChanged;
        MpiTask.StructureAlgorithm = aGp;
        MpiDifferentialEvolution tDe = new MpiDifferentialEvolution(comm);
        tDe. Iterations = 1000;
        tDe.CrossoverRate = 0.99;
        tDe.UpdatePerGeneration = 500;
        tDe.PopulationSize = 128;
        tDe.RangeMin = -1000000;
        tDe.RangeMax = 1000000;
        tDe.OnIterationChanged += onIterationChanged;
        tDe.OnPopulationConvergedToMinimum += onPopulationConvergedToMinimum;
        MpiTask.Teacher = tDe;
        Fnt fnt = new Fnt();
        if (comm.Rank == 0){
            NeuralNetwork network = fnt.FindSolutionForTask();
            fnt.SendEndMessage();
            //Work with result
        }
        else { fnt.FindSolutionForTask(); }
    }
}
```

Výpis 19: Ukázka použití knihovny

5 Testy knihovny

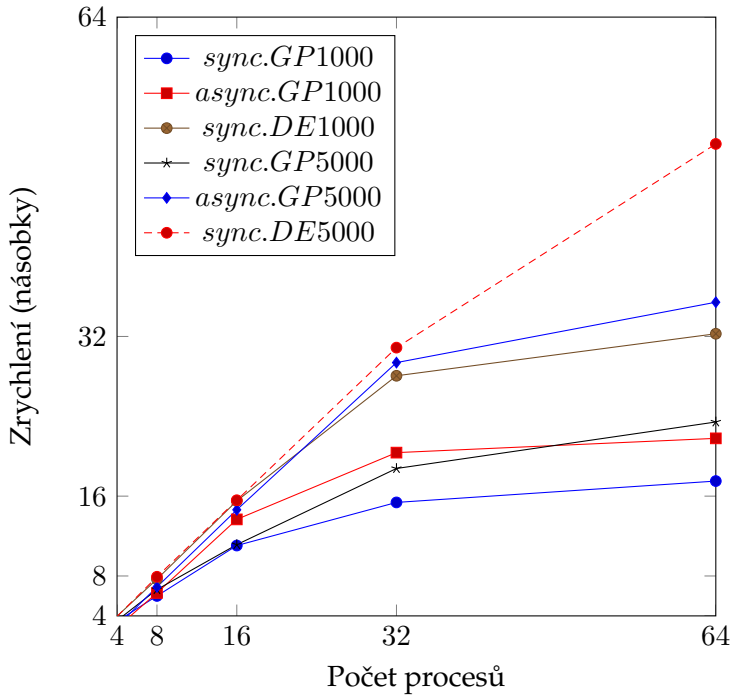
Tato kapitola pojednává o testech implementované knihovny. Funkčnost knihovny je ověřena na testech aproximace funkce sinus a předpovědi časové řady Jenkins-Box. Dále je proveden test škálovatelnosti. Knihovna byla testována, kromě testů škálovatelnosti, na počítači s procesorem Intel Core i5 4430p 3GHz a 8GB ram DDR3 1600MHz, testy využívaly všecha 4 jádra.

5.1 Testy škálovatelnosti

Pro testy škálovatelnosti byl využit superpočítač ANSELM. Tento superpočítač je složen z uzlů, testy byly prováděny na uzlech bez GPU nebo MIC akcelerátorů. Každý tento uzel obsahuje 2x osmi jádrové procesory Intel Sandy Bridge E5-2665 2,4GHz a 64GB DDR3 1600MHz operační paměti, více o specifikacích je k nalezení na stránkách `https://docs.it4i.cz/anselm-cluster-documentation/hardware-overview`.

Pro tento test byl vytvořen nový projekt pod jménem *ScalabilityTest*, který obsahuje upravené optimalizační algoritmy. Algoritmus GP optimalizující struktury byl upraven tak, že metoda *Evaluate* obsahuje tři typy výpočtu chyby populace jedinců. První typ je sériový, druhý používá synchronní MPI paralelizaci a třetí asynchronní MPI paralelizaci. Takto bylo zajištěno, že různé verze paralelizace jsou porovnány nad stejnými jedinci v populaci. Poté jsou měřeny časy vykonávání všech tří přístupů po celou dobu běhu algoritmu. Poté je spočítána efektivita paralelních přístupů oproti sériovému přístupu z výsledných časů. Algoritmus DE, optimalizující parametry neuronového stromu, je obdobně upraven jen s tím rozdílem, že zde není asynchronní přístup řešen, takže je zde jen porovnání synchronní MPI paralelizace oproti sériovému přístupu.

Test byl proveden pro různé velikosti trénovacích množin. V tabulce 2 je vidět naměřená efektivita vzhledem ke spuštěným procesům pro trénovací množinu o velikosti 1000 vzorů a v tabulce 3 pro 5000 vzorů. Tyto efektivity jsou měřeny zvlášť pro algoritmy GP a DE. Optimalizační algoritmy mají v tomto testu nastavenou velikost populace na 128 jedinců. Z tabulek vyplývá, že efektivita je lepší při použití větší trénovací množiny, pro trénovací množinu o 1000 vzorech se efektivita výrazněji snižuje u 32 spuštěných procesů, při 5000 vzorech se efektivita snižuje až u 64 procesů. Kompletní naměřená data se nachází na přiloženém CD s výpisy běhu testů. V testech FNT algoritmus běží 10 epoch a efektivita je v tabulkách zprůměrována, někdy algoritmus nestihl doběhnout omezením testu na určitý čas, proto nemusí být ve všech testech data z 10 epoch. Při běhu algoritmu jsem dále zpozoroval, že efektivita také roste s většími neuronovými stromy v populaci. V grafu na obrázku 5 je možné vidět zrychlení jednotlivých paralelních přístupů v závislosti na počtu spuštěných procesů.



Obrázek 5: Graf zrychlení optimalizačních algoritmů

Počet vzorů trénovací množiny					1000
Počet procesů	4	8	16	32	64
efektivita sync. GP	0,86	0,75	0,69	0,48	0,27
efektivita async. GP	0,71	0,79	0,85	0,63	0,34
efektivita sync. DE	0,99	0,96	0,97	0,87	0,50

Tabulka 2: Průměrné efektivity optimalizačních algoritmů (1000 vzorů)

Počet vzorů trénovací množiny					5000
Počet procesů	4	8	16	32	64
efektivita sync. GP	0,88	0,83	0,70	0,59	0,37
efektivita async. GP	0,74	0,86	0,91	0,92	0,55
efektivita sync. DE	0,98	0,99	0,97	0,97	0,80

Tabulka 3: Průměrné efektivity optimalizačních algoritmů (5000 vzorů)

5.2 Aproximace funkce sinus

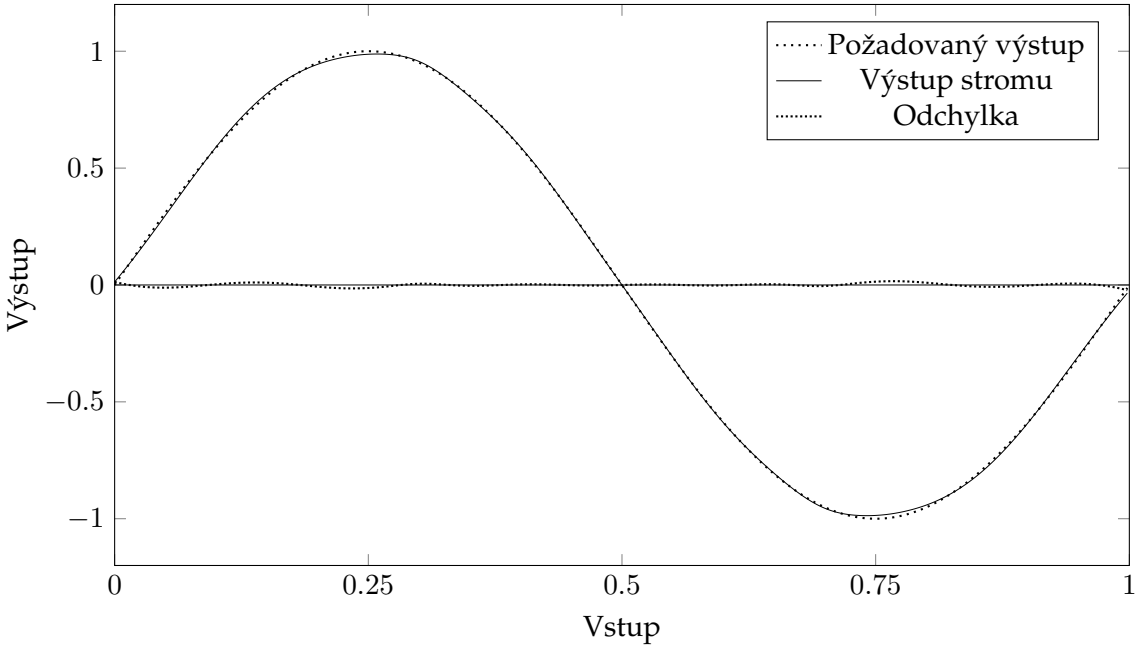
Pro tento test byl vytvořen projekt s názvem *SinusTest*. V tomto testu se hledá neuronový strom, který dokáže simulovat funkci sinus. Neuronový strom vydává výstup v intervalu $\langle -1, 1 \rangle$, vstupen do stromu jsou stupně v intervalu $\langle 0, 360 \rangle$ normalizované do reálného intervalu $\langle 0, 1 \rangle$. V tomto testu je možno nastavit počet vzorů trénovací množiny. Dobré výsledky vydával algoritmus pro nastavení z následující tabulky 4, pro toto nastavení bylo provedeno 20 testů a vybráno nejlepší řešení. Průměrná RMSE chyba výsledných neuronových stromů tohoto testu byla 0,0226.

Počet vzorů trénovací množiny	40
Parametr MaxEpochSteps třídy MpiTask	10
Parametr MaxAfterLearnIterations třídy MpiTask	2000
GP parametr EliteSize	0
GP parametr PopulationSize	256
GP parametr MaxGeneration	100
GP parametr PenaltyRate	0,005
GP parametr MutationRate	0,5
GP parametr CrossoverRate	0,5
DE parametr PopulationSize	64
DE parametr Iterations	1000
DE parametr CrossoverRate	0,99

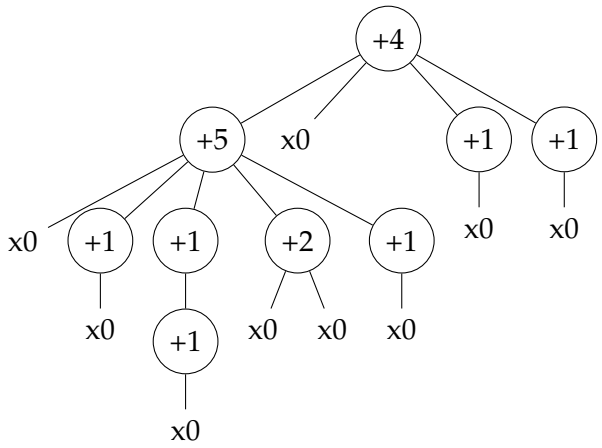
Tabulka 4: Nastavení algoritmu pro aproximaci funkce sinus

Nejlepší neuronový strom byl nalezen po 297 sekundách. Chyba neuronového stromu byla vypočítána pomocí RMSE chyby a měla hodnotu 0,00681. V grafu na obrázku 6 je možné vidět výstup neuronového stromu, skutečnou hodnotu funkce sinus a odchylku těchto hodnot. Výsledný neuronový strom bez parametrů, který má 9 neuronů, je poté možné vidět na obrázku 7.

V tomto testu bylo důležité nastavení parametrů. Při zvýšeném počtu učících iterací (nad 1000) nebo nízké hodnotě penalizace (pod 0,005) algoritmu DE docházelo k častějšímu přeučení (overfitting) neuronových stromů. Dalo se to řešit snížením počtu učících iterací nebo zvýšením parametru penalizace velikosti neuronových stromů. Výsledné neuronové stromy tak sice měli větší chybu na trénovací množině, ale zároveň měli lepší schopnost zobecňování.



Obrázek 6: Porovnání výstupu nalezeného neuronového stromu s funkcí sinus



Obrázek 7: Flexibilní neuronový strom pro aproximaci funkce sinus

5.3 Předpověď Jenkins-Box řady

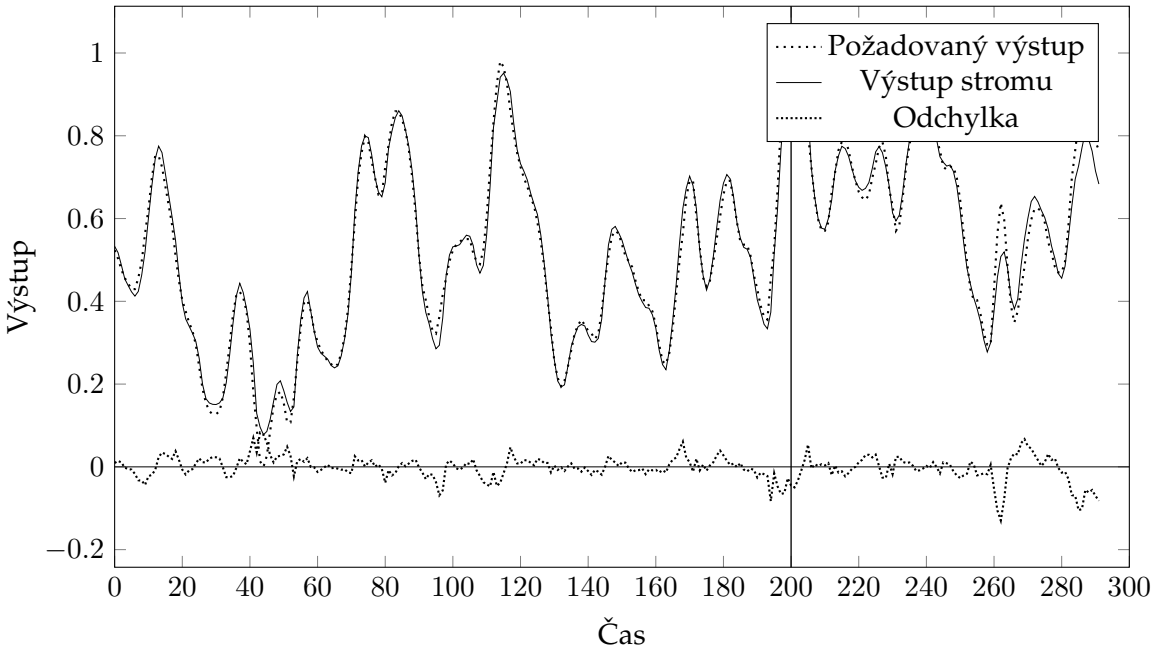
Tento test byl vybrán, protože se vyskytuje i v jiných publikacích, například je ve studii autorů Yuehui Chen a Ajith Abraham [1]. Proto bude možné porovnat tuto knihovnu. Jedná se o řadu J, která obsahuje měření průtoku plynu do plynového kotle $g(t)$ a koncentraci oxidu uhličitého (v procentech) ve výfukovém plynu proudícího z tohoto kotle $h(t)$ v čase t. Toto měření je provedeno každých 9 sekund a datový soubor obsahuje 296 takovýchto naměřených dvojic.

Trénovací množina je potom vytvořena tak, že každý vzor obsahuje jako vstup hodnoty $g(t-4)$ a $h(t-1)$ a výstup pro tento vzor je hodnota $h(t)$. Poté je tato množina rozdělena na dvě části, první část slouží k učení a jedná se o prvních 200 hodnot a zbylá část je použita k validaci. Data byla ještě normalizována do intervalu $\langle 0, 1 \rangle$.

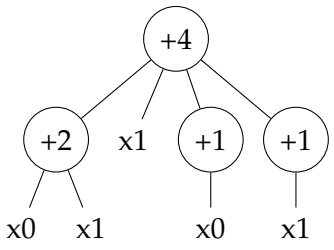
Pro tento test bylo zvoleno nastavení knihovny podle tabulky 5. Bylo provedeno 20 testů, průměrná RMSE chyba výsledných neuronových stromů pro toto nastavení byla 0,021. Průměrná RMSE chyba na validační množině byla 0,047. Nejlepší neuronový strom měl pro validační množinu RMSE chybu 0,038 a pro trénovací množinu chybu 0,019. Tento výsledný neuronový strom je možné vidět na obrázku 9 a jeho výstup porovnaný se skutečnou řadou a odchylkou je v grafu na obrázku 8.

Parametr MaxEpochSteps třídy MpiTask	10
Parametr MaxAfterLearnIterations třídy MpiTask	0
Parametr RequiredError třídy MpiTask	0,025
GP parametr EliteSize	0
GP parametr PopulationSize	256
GP parametr MaxGeneration	100
GP parametr PenaltyRate	0,01
GP parametr MutationRate	0,2
GP parametr CrossoverRate	0,2
DE parametr PopulationSize	64
DE parametr Iterations	900
DE parametr CrossoverRate	0,99

Tabulka 5: Nastavení algoritmu pro předpověď Jenkins-Box řady



Obrázek 8: Porovnání výstupu nalezeného neuronového stromu s Jenkins-Box řadou



Obrázek 9: Flexibilní neuronový strom pro předpověd Jenkins-Box časové řady

6 Závěr

Cílem této práce bylo seznámení se s neuronovou sítí typu Flexibilní Neuronový Strom a její imlementací ve formě knihovny na platformě .NET. Tato knihovna dále měla být zparalelizována pomocí MPI. Z důvodu použitého programovacího jazyka C# byla použita knihovna MPI.NET k paralelizaci. Pro optimalizaci struktury byl vybrán algoritmus Genetické Programování a byl zparalelizován synchronním i asynchronním přístupem pro porovnání. Pro optimalizaci parametrů byl vybrán algoritmus Diferenciální Evoluce, který byl zparalelizován synchronním přístupem. Knihovna byla navržena tak, aby ji bylo možné rozšířit i o jiné optimalizační algoritmy.

Knihovna byla testována na problému aproximace funkce sinus. Tento test byl proveden mnohokrát pro různé hodnoty parametrů optimalizačních algoritmů a nejlepší vyzkoušené hodnoty parametrů jsou k nalezení v tabulce 7. Jako test předpovědi časové řady byla vybrána data Jenkins-Box. Z mnoha provedených testů byla průměrná chyba na validační množině vyšší než na trénovací množině (0,021 vs 0,047). Neuronové stromy byly lehce přeučeny oproti publikaci autorů Yuehui Chen a Ajith Abraham [1], kde jejich neuronový strom měl chybu na validační množině podobnou s chybou na trénovací množině. Problém přeučení by se možná dal řešit zvolením jiných parametrů nebo doimlementováním techniky předčasného ukončení učení neuronového stromu, kdy by trénovací množina byla rozdělena ještě na část testovací. A pokud by se chyba na této testovací množině začala zvyšovat, tak by se zakázalo učení tohoto neuronového stromu.

Dále byl proveden test škálovatelnosti navržené knihovny na superpočítači Anselm. V tomto testu dosahoval algoritmus Diferenciální Evoluce slušné efektivity i při použití více jader, tato efektivita ale závisí na počtu vzorů trénovací množiny. Alogitmus Genetického Programování s asynchronní MPI komunikací dosahoval lepších výsledků než jeho synchronní verze při použití 8 a více jader. Tato asynchronní verze dále měla výhodu u populace neuronových stromů rozdílných velikostí. Efektivita paralelních verzí těchto algoritmů by mohla být dále zvýšena využitím hybridního přístupu, kdy by nebyl spuštěn každý proces na jednom jádře, ale tento proces by běžel na jednom procesoru a všechna jádra tohoto procesoru by byla využita pomocí vláken. Toto by mohlo snížit vliv komunikace mezi procesy na efektivitu, kdy by bylo použito mnoho výpočetních uzlů.

Knihovna by mohla být v budoucnu přepsána do jazyka C++ z důvodu lepšího výkonu s využitím MPI knihovny od Intelu, která je nainstalována na superpočítači Anselm. Dále by mohla být knihovna použita na reálné problémy s většími trénovacími množinami, které by byly bez paralelizace časově náročné.

7 Reference

[1] Yuehui Chen, Ajith Abraham, *Tree-Structure based Hybrid Computational Intelligence: Theoretical Foundations and Applications,* Springer, 2010

[2] Pavel Piskoř, *Framework pro neuronovou síť Flexible Neural Tree*, Ostrava, 2013. 72 s. Diplomová práce na fakultě elektrotechniky a informatiky Vysoké školy báňské na katedře informatiky. Vedoucí diplomové práce doc. Mgr. Jiří Dvorský, Ph.D.

[3] Byoung-Tak Zhang, Peter Ohm, Heinz Mühlenbein, *Evolutionary Induction of Sparse Neural Trees,* MIT Press, 1997

[4] Xuejiao Lei, Yuehui Chen, *Multiclass Classification of Microarray Data Samples with Flexible Neural Tree,* IEEE, 2012

[5] Lizhi Peng, Bo Yang, Lei Zhang, Yuehui Chen, *A parallel evolving algorithm for flexible neural tree,* Elsevier Science Publishers B. V., 2011

[6] Yuehui Chen, Feng Chen, Jack Y. Yang, *Evolving MIMO Flexible Neural Trees for Nonlinear System Identification,* CSREA Press, 2007

[7] Yuehui Chen, Lizhi Peng, Ajith Abraham, *Gene Expression Profiling Using Flexible Neural Trees,* Springer Berlin Heidelberg, 2006

[8] Ivo Vondrák, *Umělá inteligence a neuronové sítě,* VŠB-TU Ostrava, 2001

[9] Ivan Zelinka, Zuzana Oplatková, Miloš Šeda, Pavel Ošmera, František Včelař, *Evoluční výpočetní techniky: Principy a aplikace,* BEN - technická literatura, 2009

[10] G. Bard Ermentrout, David H. Terman, *Mathematical Foundations of Neuroscience,* Springer Science & Business Media, 2010

[11] E. Massio Grimaldi, F. Grimaccia, M. Mussetta, R. E. Zich, *PSO as an effective learning algorithm for neural network applications,* IEEE, 2004

[12] Jeremiah Willcock, Andrew Lumsdaine, Arch Robison, *Using MPI with C# and the Common Language Infrastructure,* ACM, 2002

[13] Randall S. Sexton, Robert E. Dorsey, John D. Johnson, *Beyond Back Propagation: Using Simulated Annealing for Training Neural Networks,* IGI Global, 1999

[14] Ding-Jun Chen, Chung-Yeol Lee, Cheol-Hoon Park, Pedro Mendes, *Parallelizing Simulated Annealing Algorithms Based on High-performance Computer,* Kluwer Academic Publishers, 2007

[15] Tobias Blickle, Lothar Thiele, *A Comparison of Selection Schemes used in Genetic Algorithms,* Evolutionary Computation, 1997

[16] Abdual-Salam, Abdul-Kader, Abdel-Wahed, *Comparative study between Differential Evolution and Particle Swarm Optimization algorithms in training of feed-forward neural network for stock price prediction*, IEEE, 2010

[17] Indiana University, *MPI.NET : High-Performance C# Library for Message Passing*, Dostupné z WWW: *http://www.osl.iu.edu/research/mpi.net/*, 28.2.2015

[18] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard : Version 2.2*, Dostupné z WWW: *http://www. mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf*, 28.2.2015